7. Gaia
Haskell (Programazio Funtzionala)

7.1. Sarrera

7.1.1. Helburua

Programazio funtzionalaren (edo funtzioen bidezko programazioaren) ezaugarriak aztertzea da gai honetako helburua. Horretarako Haskell lengoaia erabiliko da.

Lengoaia funtzional bat (edo funtzioetan oinarritutako lengoaia bat) ikasteak errekurtsibitatearen teknikan sakontzea dakar alde batetik. Errekurtsibitatearen barruan, hasteko, oinarrizko errekurtsibitatea berrikusiko da eta gero murgilketa izenarekin ezagutzen den teknika landuko da. Beste aldetik, zerrenden eraketa izenarekin ezagutzen den baliabidea, sarrera/irteera eta fitxategien oinarrizko erabilera ere landuko dira.

7.1.2. Haskell

Haskell programazio-lengoaia funtzionala da, hau da, funtzioetan oinarritutako programazio-lengoaia da.

Haskell lengoaian programa bat funtzio multzo bat izan ohi da. Funtzio bakoitza era independentean exekuta daiteke eta baita beste funtzio batetik deituz ere (kasu bietan parametro formalen ordez argumentu errealak ipiniz).

Haskell-en web orri ofiziala http://haskell.org da eta bertatik hainbat sistema eragilerentzat Haskell plataforma instala daiteke. Haskell erabiltzen hasteko era errazena WinHugs interpretatzailea da eta http://www.haskell.org/hugs/ helbidetik eraitsi daiteke. GHC konpilatzailea ere eskura daiteke http://www.haskell.org/ghc/ helbidean.

7.2. Oharrak

7.2.1. Maiuskula eta minuskulen erabilera izenetan

Maiuskulak eta minuskulak ondo erabiltzeko honako arau hauek jarraitu behar dira:

- Datu-moten izenak maiuskulaz hasten dira: Int, Bool, Char, …
- Datu-mota berriak definitzean, eragiketa eraikitzaileen izenek maiuskulaz hasi behar dute: Phutsa, Pilaratu, Ahutsa, Eraiki, …
- Moten ordez ipintzen diren aldagaien izenak minuskulaz hasten dira: t.
- True eta False konstanteak maiuskulaz hasten dira.
- Eraikitzaileak ez diren beste eragile eta funtzio denak eta aldagaien izenak minuskulaz hasi behar dute (maiuskulak erdian ipintzea egon arren): bikoitia, bakoitia, leh, hond, badago, luzera, x, s, r, p, q, zatb, kenduLuz, kenduLuzBik, …

7.2.2. Azalpenak ipintzeko bi era

Haskell-ez idatzitako programetan azalpenak ipini nahi izanez gero, bi aukera daude:

- -- erabiliz:

  -- sinboloen atzetik lerroa bukatu arte datorren dena azalpena izango da.

  Adibidea:

  --Hau azalpen bat da eta lerroa bukatzen denean bukatuko da azalpena

- {- eta -} erabiliz:

  {- eta -} sinboloen artean doan dena azalpena da. Lerro batetik bestera pasatuta ere azalpenak aurrera jarraitzen du.

  Adibidea:

  ```
  {- Hau azalpena da baina
  lerroz alda gaitezke. -}
  ```

7.3. Haskell-en aurredefinituta dauden oinarrizko motak

7.3.1. Bool

- Balioak: {True, False}
- Eragiketak:
  - ✓ && (eta, konjuntzioa, and)
  - ✓ || (edo, disjuntzioa, or)
  - ✓ not (ez, ukapena, not)
- Propietateak (φ, ψ eta γ espresio boolearrak direla kontsideratuz):
  - ✓ φ && (ψ && γ) ≡ (φ && ψ) && γ (&& elkarkorra da)
  - ✓ φ || (ψ || γ) ≡ (φ || ψ) || γ (|| elkarkorra da)
  - ✓ φ && ψ ≡ ψ && φ (&& trukakorra da)
  - ✓ φ || ψ ≡ ψ || φ (|| trukakorra da)
  - ✓ φ && (ψ || γ) ≡ (φ && ψ) || (φ && γ) (&& banakorra da || eragilearekiko)
  - ✓ φ || (ψ && γ) ≡ (φ || ψ) && (φ || γ) (|| banakorra da && eragilearekiko)
  - ✓ not(φ && ψ) ≡ (not φ) || (not ψ) (De Morgan)
  - ✓ not(φ || ψ) ≡ (not φ) && (not ψ) (De Morgan)
  - ✓ φ || φ ≡ φ (Idenpotentzia)
  - ✓ φ && φ ≡ φ (Idenpotentzia)
  - ✓ φ && (φ || ψ) ≡ φ (Absortzioa)
  - ✓ φ || (φ && ψ) ≡ φ (Absortzioa)
  - ✓ True && φ ≡ φ (True elementu neutroa da && eragilearentzat)
  - ✓ False || φ ≡ φ (False elementu neutroa da || eragilearentzat)
  - ✓ False && φ ≡ False (False elementu nulua da && eragilearentzat)
  - ✓ True || φ ≡ True (True elementu nulua da || eragilearentzat)
  - ✓ not True ≡ False
  - ✓ not False ≡ True
  - ✓ not (not φ) ≡ φ (ukapen bikoitza)
  - ✓ φ || not φ ≡ True (hirugarren aukerarik ez)
  - ✓ φ && not φ ≡ False (kontraesana)

  Hor, φ, ψ eta γ espresio boolearrak adierazteko erabili dira.

7.3.2. Int

- Balioak: tarte mugatu bateko zenbaki osoak (minBound::Int, maxBound::Int)
- Eragile aritmetikoak:
  - ✓ - (zeinu negatiboa)
  - ✓ + (batuketa)
  - ✓ – (kenketa)

- ✓ * (biderketa)
- ✓ ^ (zenbaki osoen arteko berreketa; 2 ^ 3 = 8)
- ✓ div (zatiketa osoa; idazteko bi era: 16 `div` 3 = 5 edo div 16 3 = 5, eta ` karakterea P letraren ondoan dagoen azentu-marka da, hau da, azentu kamutsa; 6 `div` 4 = 1; 8 `div` 2 = 4)
- ✓ mod (zatiketa osoaren hondarra; idazteko bi era: 16 `mod` 3 = 1 edo mod 16 3 = 1, eta ` karakterea P letraren ondoan dagoen azentu-marka da, hau da, azentu kamutsa; 6 `mod` 4 = 2; 8 `mod` 2 = 0)
- ✓ / (zatiketa erreala; 16 / 3 = 5.333; 6 / 4 = 1.5; 8 / 2 = 4.0)
- ✓ abs (balio absolutua; abs (-7) = 7)
- ✓ negate (zeinu aldaketa; negate (-5) = 5; negate 5 = -5)
- ✓ signum (1, 0 edo -1 itzuliko du zenbakia positiboa, zero edo negatiboa al den adierazteko)

+, -, * eta ^ eragileak era prefixuan edo infixuan erabil daitezke. Era prefixuan erabiltzeko, parentesien artean ipini behar dira:

6 + 4 = 10 (+) 6 4 = 10

- Propietateak:
  - ✓ x + (y + z) = (x + y) + z — + elkarkorra da
  - ✓ x * (y * z) = (x * y) * z — * elkarkorra da
  - ✓ x + y = y + x — + trukakorra da
  - ✓ x * y = y * x — * trukakorra da
  - ✓ -(x + y) = (-x) + (-y)
  - ✓ x * (y + z) = (x * y) + (x * z) — * banakorra da + eragilearekiko
  - ✓ 0 + x = x — 0 elementu neutroa da + eragilearentzat
  - ✓ 1 * x = x — 1 elementu neutroa da * eragilearentzat
  - ✓ 0 * x = x — 0 elementu nulua da * eragilearentzat
  - ✓ -(-x) = x
  - ✓ (-x) + x = 0
  - ✓ abs (abs x) = abs x
  - ✓ …

7.3.3. Integer

- Balioak: zenbaki osoak inolako mugarik gabe
- Eragile aritmetikoak: Int kasukoak
- Propietateak: Int kasukoak

7.3.4. Float, Double

- Balioak: zenbaki errealak
- Eragile aritmetikoak:

- ✓ Int motarentzat aipatutako denak div eta mod izan ezik.
- ✓ ^ (berretzaileak osoa izan behar du; (2.6) ^ 3 onartzen da baina (2.6) ^ (3.4) ez)
- ✓ ** (zenbaki errealen arteko berreketa; (2.5) ** (3.1) = 17.12434728726902)
- ✓ truncate (puntuaren ondorengoa ezabatzeko; truncate 6.4 = 6)

** eragilea era prefixuan edo infixuan erabil daiteke. Era prefixuan erabiltzeko, parentesien artean ipini behar da:

```
(2.5) ** (3.1) = 17.12434728726902
(**) (2.5) (3.1) = 17.12434728726902
```

- Propietateak: Int kasukoen antzekoak

7.3.5. Char

- Balioak: karaktereak ('a', …, 'z', 'A', …, 'Z', '0', …, '9', '$', '?', …) 0ren teklaren ondoan dagoen teklako apostrofoa erabiliz
- Eragileak:
  - ✓ succ: (hurrengo karakterea; succ 'B' = 'C')
  - ✓ pred: (aurreko karakterea; pred 'B' = 'A')
- Propietateak:
  - ✓ succ(pred x) = x
  - ✓ pred(succ x) = x

Zenbakiak erabiliz ere adieraz daitezke karaktereak. Adibidez, 'A' karakterea '\65' moduan adieraz daiteke, 'a' karakterea '\97' moduan, '0' karakterea '\48' moduan eta abar. '\0' espresiotik '\127' espresiora doazen kaktereak ASCII karaktereak dira.

7.3.6. Oinarrizko motentzako eragile erlazionalak

Jarraian erakusten diren eragile erlazionalak aipatu diren oinarrizko mota denentzat balio dute:

- ✓ == (berdin)
- ✓ /= (ezberdin)
- ✓ > (handiagoa)
- ✓ >= (handiagoa edo berdina)
- ✓ < (txikiagoa)
- ✓ <= (txikiagoa edo berdina)

7.3.7. Oinarrizko motentzat errekurtsiboak ez diren eragiketa berrien definizioa

Lehenago aipatu diren eragiketak erabiltzeaz gain eragiketa berriak ere defini eta erabil daitezke.

Eragiketak definitzean mota eta ekuazioak emango ditugu. Ekuazio bakoitzari zenbaki bat egokituko diogu geroago ekuazio hori aipatu ahal izateko.

Funtzio berrien motak ematean, hasteko, datuen motak ipiniko dira -> geziaz bereizita eta bukatzeko emaitzaren mota ipiniko da, hau ere datuen motetatik -> geziaz bereizita. Beti emaitza bakarra dago. Beraz azkeneko mota emaitzari dagokiona izango da eta aurreko denak datuei dagozkienak izango dira. Gerta daiteke daturik ez egotea, datu bakarra egotea edo datu bat baino gehiago egotea, baina beti emaitza bakarra egongo da. Emaitza bakarra dagoenean, emaitza bakar horren mota eman beharko da besterik gabe.

Bost adibide emango dira jarraian:

- **pi2**: funtzio hau funtzio konstantea da, izan ere ez du daturik behar eta beti 3.1415 zenbaki erreala itzuliko du.

  Eragiketaren **mota**:
  ```
  pi2:: Float
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  pi2 = 3.1415
  ```

  Oharra: funtzio honi pi2 deitu zaio Haskell-en **pi** funtzioa aurredefinituta dagoelako.

- **f**: sarrerako datuak onartu arren, beti balio bera itzuliko duten funtzioak ere defini daitezke.

  Eragiketaren **mota**:
  ```
  f:: Int -> Int
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  f x = 100
  ```

  Definizioaren arabera, f funtzioari edozein zenbaki oso emanda (x), funtzioak beti 100 balioa itzuliko du.

- **bikoitia**: zenbaki oso bat emanda, True itzuliko du bikoitia baldin bada eta False bakoitia baldin bada.

  Eragiketaren **mota**:
  ```
  bikoitia:: Int -> Bool
  ```
  Eragiketa definitzen duen **ekuazioa**:
  ```
  bikoitia x = (x `mod` 2) == 0
  ```

  bikoitia funtzioak x balioarentzat x `mod` 2 eta 0 konparatzearen balioa itzuliko duela adierazten da ekuazio horren bidez (konparazioaren emaitza True edo False izango da).

  Eragiketa bera definitzeko beste aukera bat honako hau izango litzateke:
  ```
  bikoitia x
      | (x `mod` 2) == 0     = True      (1)
      | otherwise            = False     (2)
  ```

  Definizio hori honela ulertu beharko genuke:

x `mod` 2 eta 0 berdinak baldin badira, True itzuli eta bestela False itzuli.

- **bakoitia**: zenbaki oso bat emanda, True itzuliko du zenbaki hori bakoitia baldin bada eta False itzuliko du zenbaki hori bikoitia baldin bada.

  Eragiketaren **mota**:
  bikoitia:: Int -> Bool
  Eragiketa definitzen duen **ekuazioa**:
  bakoitia x = not(bikoitia x) (1)

  Kasu honetan, x zenbaki oso bat emanda, *bakoitia* funtzioak x zenbaki horrentzat itzuliko duena *bikoitia* funtzioak x zenbaki horrentzat itzuliko lukeenaren kontrakoa dela adierazten da. Beraz, hasteko *bikoitia* funtzioak x balioarentzat itzultzen duena kalkulatuko da eta gero balio horren aurkakoa hartuko da *not* eragilearen bidez.

  Adibide honetan ikusten den bezala, funtzio berri bat definitzean, aurretik definituta dauden funtzioak ere erabil daitezke.

- **hand3**: hiru zenbaki oso emanda handiena itzuliko du.

  Eragiketaren **mota**:
  hand3:: Int -> Int -> Int -> Int
  Eragiketa definitzen duten **ekuazioak**:

```
hand3 x y z
    | x >= y && x >= z      = x      (1)
    | y > x && y >= z       = y      (2)
    | otherwise             = z      (3)
```

  Definizio hori honela ulertu beharko genuke:
  x >= y && x >= z betetzen bada, orduan x balioa itzuli, bestela y > x && y >= z betetzen bada, orduan y itzuli eta bestela z itzuli.

  Adibide honetan bezala baldintza-zerrenda bat agertzen denean, baldintzak goitik behera aztertuko dira Haskell-en eta betetzen den lehenengo baldintzaren kasuari dagokion emaitza itzuliko da.

7.3.8. Oinarrizko errekurtsioa zenbaki osoentzat

Zenbaki osoen gaineko eragiketak burutzen dituzten funtzio errekurtsibo batzuk azalduko dira atal honetan.

- **bider**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatuko du x balioa y aldiz batuz. Kasu berezi bezala, y-ren balioa negatiboa denean errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

```
bider:: Int -> Int -> Int
```

Eragiketa definitzen duten **ekuazioak**:

```
bider x y
  | y < 0              = error "Bigarren balioa negatiboa da."
  | x == 0 || y == 0   = 0
  | x == 1             = y
  | otherwise          = x + bider x (y – 1)
```

- **bneg**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatuko du. Bigarren balioa negatiboa denean ere emaitza ondo kalkulatuko du. Funtzio honen definizioan, aurretik definitu den *bider* funtzioa erabiliko da. Ekuazioen bidezko definizioan ikus daitekeen bezala, *bneg* funtzioa ez da errekurtsiboa, ez baitio bere buruari deitzen. Errekurtsibitatea *bider* funtzioan dago.

  Eragiketaren **mota**:

  ```
  bneg:: Int -> Int -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  bneg x y
    | x == 0 || y == 0          = 0
    | (x < 0) && (y < 0)        = bider (-x) (-y)
    | (x > 0) && (y > 0)        = bider x y
    | (x < 0) && (y > 0)        = bider x y
    | (x > 0) && (y < 0)        = bider (-x) (-y)
  ```

- **errusiar**: funtzio honek, x eta y zenbaki osoak emanda, bien arteko biderkadura kalkulatuko du *biderketa errusiarra* bezala ezagutzen den metodoari jarraituz. Funtzio honen definizioan, aurretik definitu den *bikoitia* funtzioa erabiliko da. Kasu berezi bezala, y-ren balioa negatiboa denean, errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

  ```
  errusiar:: Int -> Int -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  errusiar x y
    | y < 0                     = error "Bigarren balioa negatiboa da."
    | y == 0                    = 0
    | bikoitia y                = errusiar (x + x) (y `div` 2)
    | otherwise                 = x + errusiar (x + x) (y `div` 2)
  ```

- **zatios**: funtzio honek, x eta y zenbaki osoak emanda, x eta y-ren arteko zatidura osoa kalkulatuko du, horretarako x balioari y balioa zenbat aldiz kendu ahal zaion zenbatuz. Kasu berezi bezala, y-ren balioa zero denean errore-mezua aurkeztuko du. Gainera, x edota y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

  ```
  zatios:: Int -> Int -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  zatios x y
  ```

```
  | y == 0              = error "Zatitzailea 0."
  | (x < 0) || (y < 0)  = error "Gutxienez bietako bat negatiboa da."
  | x < y               = 0
  | otherwise           = 1 + zatios (x - y) y
```

- **zatihond**: funtzio honek, x eta y zenbaki osoak emanda, x eta y-ren arteko zatiketa osoaren hondarra kalkulatuko du. Horretarako, x balioari eta ondoren lortuko diren kendurei y balioa kenduz joango da, y baino txikiagoa den balio bat gelditu arte. Kasu berezi bezala, y-ren balioa zero denean errore-mezua aurkeztuko du. Gainera, x edota y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

  Eragiketaren **mota**:

  ```
  zatihond:: Int -> Int -> Int
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  zatihond x y
    | y == 0              = error "Zatitzailea 0."
    | (x < 0) || (y < 0)  = error "Gutxienez bietako bat negatiboa da."
    | x < y               = x
    | otherwise           = zatihond (x - y) y
  ```

7.4. Zerrendak

Zerrenda bat honako era honetan adieraz daiteke: [2, 7, 35, -8, 0]
Zerrenda hori zenbaki osoz osatuta dago eta bere mota [Int] da. Zerrenda hutsa honako era honetan adierazten da: []

Zerrenden mota parametrodun mota da, [t], eta t parametroaren arabera balio boolearrez osatutako zerrendak, zenbaki osoz osatutako zerrendak, karakterez osatutako zerrendak eta abar eduki ditzakegu. Beraz, aurretik definitutako edozein mota erabiliz mota horretako elementuz osatutako zerrendak defini daitezke baina zerrenda bateko elementu denak mota berekoak izan behar dute.

Adibideak:

[True, True, True, False, True]
Bere mota [Bool] da (boolearrez osatutako zerrenda).

[[0, 5], [], [77, -50, 3]]
Bere mota [[Int]] da (osoez osatutako zerrendez eratutako zerrenda)

['d', 'd', 'a', 'b', 'z', '!', 'g']
Bere mota [Char] da (karakterez eratutako zerrenda)

7.4.1. Eragiketa eraikitzaileak zerrendentzat

Datu-mota bati dagozkion eragiketa eraikitzaileak mota horretako balioak zein diren adierazteko dira.

Edozein zerrenda honako bi eragile hauek erabiliz eraiki daiteke:

- [] **Zerrenda hutsa sortu**
- (:) **Ezkerretik elementu bat erantsi**

Lehenengo eragiketak zerrenda hutsa sortzeko balio du eta bigarrenak ezkerretik elementuak banan-banan erantsiz edo sartuz joateko balio du.

Esate baterako [6, 10] zerrenda honako era honetan eraikiko litzateke:

(:) 6 ((:) 10 [])

(:) funtzioa era infixuan erabil daiteke eta, gainera, ohikoena era infixuan erabiltzea da. Era infixua erabiliz [6, 10] zerrenda honela adieraziko genuke:

6:10:[]

Espresio horren esanahia honako hau da: Hasteko zerrenda hutsa sortuko litzateke, gero 10 zenbakia gehituko litzaioke zerrenda hutsari eta bukatzeko 6 balioa gehituko litzaioke dagoeneko 10 zenbakia duen zerrendari.

[] → [10] → [6, 10]

[2, 7, 35, -8, 0] zerrenda honako era honetan eraikiko litzateke: 2:7:35:-8:0:[]

Haskell lengoaian [6,10] eta 6:10:[] baliokideak dira, biak onartzen dira, baina ekuazioen bidezko espezifikazioaren bidez eragiketa berriak definitzeko eta eragiketa berri horien propietateak frogatzeko bigarren aukera egokiagoa da.

Edozein zerrenda, jarraian aipatzen diren bi egitura hauetakoren batera egokitzen da:

- ✓ [] (zerrenda hutsa da)
- ✓ x:s (ez da zerrenda hutsa eta bere lehenengo elementua x da eta beste elementu denek s azpizerrenda osatzen dute).

[2, 7, 35, -8, 0] zerrenda 2:7:35:-8:0:[] eran idatz daiteke
x: s
s azpizerrenda 7:35:-8:0:[] zerrenda da

Zerrenda batek bi elementu edo gehiago dituenean
x:z:s
erakoa dela ere esan daiteke, komeni bada. Lehenengo elementua **x** izango da, bigarrena **z** eta gainontzeko elementuek **s** azpizerrenda osatzen dute.

[2, 7, 35, -8, 0] zerrenda 2:7:35:-8:0:[] eran idatz daiteke
x:z:s

Eragiketa eraikitzaileen motak honako hauek dira:

```
[] :: [t]
(:) :: t -> [t] -> [t]
```

Lehenengo eragiketaren motak, hau da, [] eragiketaren motak, eragiketa horrek geuk nahi dugun motako (t motako) zerrenda hutsa sortuko duela adierazten du. [] funtzioak ez du daturik behar eta [t] motako emaitza sortuko du.

Bigarren eragiketaren kasuan, (:) eragileari t motako elementu bat eta t motako zerrenda bat emanda, t motako zerrenda bat itzuliko duela adierazten da. Beraz, (:) funtzioak bi datu jasoko ditu eta emaitza bat itzuliko du. Azkeneko mota emaitzari dagokiona da eta aurretik dauden motak datuei dagozkien motak dira.

Eragiketen motak eta elementuen motak ematean **::** notazioa erabiltzen da. [] eta (:) funtzioen mota ematean ikusten da funtzioetan, hasteko, datuen motak ipiniko direla -> geziaz bereizita eta bukatzeko emaitzaren mota ipiniko dela, hau ere datuen motetatik -> geziaz bereizita. Sarrerako daturik ez badago, emaitzaren mota bakarrik emango da, [] eragilearen kasuan bezala. Funtzioen kasuan gerta daiteke sarrerako daturik ez egotea, datu bakarra egotea edo datu bat baino gehiago egotea, baina beti emaitza bat egongo da.

Haskell lengoaiaren interpretatzailean funtzioen motari buruz galde dezakegu. Esate baterako, [] funtzioaren mota zein den jakiteko honako hau idatzi beharko genuke:

```
:type []
```

Erantzuna honako hau izango da:

```
[] :: [t]
```

(:) funtzioaren mota zein den jakiteko honako hau idatzi beharko genuke:

```
:type (:)
```

Eta kasu honetan erantzuna honako hau izango litzateke:

```
(:) :: a -> [a] -> [a]
```

Espresio horretan `a` edozein mota adierazteko erabili da.

Edozein mota edo mota ezezagunak adierazteko Haskell-ek letra minuskulak erabiltzen ditu: `a`, `b`, `c`, …, `t`, …

7.4.2. Errekurtsiboak ez diren eragiketa batzuk zerrendentzat

Atal honetan zerrendekin kalkuluak burutzeko balio duten eta errekurtsiboak ez diren eragiketa batzuk definituko dira. Funtzio bat edo eragiketa bat definitzeko, funtzioaren mota eta funtzioak zer egiten duen zehazten duten ekuazioak eman behar dira.

- **leh**: Zerrenda bat emanda, zerrendako lehenengo elementua itzuliko du. Zerrenda hutsa baldin bada, errorea sortuko da.

  Adibideak:
  leh [4, 7, 8] = 4
  leh [False, True] = False
  leh [[0, 5], [], [77, -50, 3]] = [0, 5]

  Eragiketaren **mota**:
  leh:: [t] -> t

  Eragiketa definitzen duten **ekuazioak**:
  leh [] = error "Zerrenda hutsa. Ez dago lehenengo elementurik." (1)
  leh (x:s) = x (2)

  Errore-mezu bat aurkezteko Haskell-eko *error* funtzioa erabiltzen da eta zein mezu aurkeztu nahi den zehaztu beharko da.

- **hond**: Zerrenda bat emanda, zerrendako lehenengo elementua kenduz lortzen den zerrenda itzuliko du. Zerrenda hutsa baldin bada, errorea sortuko da.

  Adibideak:
  hond [4, 7, 8] = [7, 8]

```
hond [False, True] = [True]
hond [[0, 5], [], [77, -50, 3]] = [[], [77, -50, 3]]
```

Eragiketaren **mota**:

```
hond:: [t] -> [t]
```

Eragiketa definitzen duten **ekuazioak**:

```
hond [] = error "Zerrenda hutsa. Ez dago hondarrik."   (1)
hond (x:s) = s                                         (2)
```

Zerrenda hutsari dagokion errore-mezua aurkezteko Haskell-eko *error* funtzioa erabili da eta aurkeztu nahi den mezua zein den zehaztu da komatxoen artean.

- **hutsa_da**: Zerrenda bat emanda, True itzuliko du zerrenda hutsa baldin bada eta False itzuliko du zerrenda hutsa ez bada.

  Adibideak:

  ```
  hutsa_da [] = True
  hutsa_da [4, 7, 8] = False
  hutsa_da [False, True] = False
  ```

  Eragiketaren **mota**:

  ```
  hutsa_da:: [t] -> Bool
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  hutsa_da [] = True          (1)
  hutsa_da (x:s) = False      (2)
  ```

  Emandako zerrenda hutsa baldin bada (hau da, [] erakoa baldin bada), orduan True itzuliko da. Emandako zerrenda hutsa ez bada (hau da, x:s erakoa baldin bada, x zerrendako lehenengo elementua izanda eta s zerrendako gainontzeko elementuez osatutako azpizerrenda izanda), orduan False itzuliko da.

7.4.3. Errekurtsiboak diren eragiketa batzuk zerrendentzat

Atal honetan errekurtsibitatea duten funtzio batzuk definituko dira zerrendentzat.

- **badago**: t motako elementu bat eta t motako zerrenda bat emanda, True balioa itzuliko du elementua zerrendan baldin badago eta False itzuliko du ez badago.

  Adibideak:

  ```
  badago 8 [] = False
  badago 8 [4, 8, 7, 8] = True
  badago False [False, True, True] = True
  badago [10, 9] [[0, 5], [], [77, -50, 3]] = False
  ```

  Eragiketaren **mota**:

  ```
  badago:: Eq t => t -> [t] -> Bool
  ```

  Eragiketa definitzen duten **ekuazioak**:

  ```
  badago x [] = False                    (1)
  badago x (y:s)
      | x == y      = True               (2)
      | x /= y      = badago x s         (3)
  ```

  Ekuazio horien bitartez x elementua zerrenda batean agertzen al den ala ez erabakiko da. Zerrenda hutsa baldin bada (hau da, [] erakoa baldin bada), x ez da hor egongo eta False itzuliko da zuzenean. Zerrenda hutsa ez bada, (hau da, y:s erakoa baldin bada, zerrendako lehenengo elementua y dela eta zerrendako beste elementu denez osatutako azpizerrenda s dela kontsideratuz) x balioa y:s zerrendako lehenengo elementuaren berdina baldin bada (hau da, x balioa y-ren berdina baldin bada) erantzuna True izango da zuzenean baina x balioa y:s zerrendako lehenengo elementuaren berdina ez bada (hau da x eta y berdinak ez badira), orduan x elementua s azpizerrendan ba al dagoen begiratu beharko da.

  Oharra: 'badago' funtzioaren definizioan edozein t mota onartzen da. Definizio horretan ikusten da 'badago' funtzioak t motako bi elementu berdinak al diren erabaki beharko duela. Ondorioz, berdina izatea edo berdina ez izatea definituta duten motentzat bakarrik balio du 'badago' funtzioak. Hori adierazteko, Eq t => ipini behar da funtzioaren mota zehazterakoan.

- **(++)**: t motako bi zerrenda emanda, bata besteari erantsiz edo biak elkartuz lortzen den zerrenda itzuliko du.

  Adibideak:

  ```
  [] ++ [] = []
  [4, 8, 7, 8] ++ [] = [4, 8, 7, 8]
  [1, 8] ++ [4, 8, 7] = [1, 8, 4, 8, 7]
  [False, True, True] ++ [False, False] = [False, True, True, False, False]
  ```

Funtzio honen izena berez (++) da eta [1, 8] ++ [4, 8, 7] idatzi beharrean beste era honetara ere idatz daiteke:

(++) [1,8] [4,8,7]

Hala ere, ohikoena [1, 8] ++ [4, 8, 7] idaztea da. Beraz, kanpoan ipintzen bada, parentesiekin ipini behar da eta zerrenden artean ipintzen bada, parentesirik gabe ipini behar da.

Eragiketaren **mota**:

```
(++):: [t] -> [t] -> [t]
```

Eragiketa definitzen duten **ekuazioak**:

```
[] ++ s = s                    (1)
(x:r) ++ s = x:(r ++ s)        (2)
```

Bigarren ekuazioak x:r eta s zerrendak elkartzeko lehenengo r eta s zerrendak elkartu eta gero r ++ s zerrenda berriari x elementua ezkerretik gehitu behar zaiola dio.

- **luzera**: t motako zerrenda bat emanda, elementu kopurua (zerrenda horretan zenbat elementu dauden) itzuliko du.

Adibideak:

```
luzera [] = 0
luzera [5, 8, 7, 8] = 4
luzera [False, True, True] = 3
luzera [[0, 5], [], [77, -50, 3]] = 3
```

Eragiketaren **mota**:

```
luzera:: [t] -> Int
```

Eragiketa definitzen duten **ekuazioak**:

```
luzera [] = 0                  (1)
luzera (x:r) = 1 + (luzera r)  (2)
```

Emandako zerrenda hutsa baldin bada (hau da, [] erakoa baldin bada), orduan luzera edo elementu kopurua 0 da. Emandako zerrenda hutsa ez bada (hau da, x:r erakoa baldin bada), zerrendak gutxienez x elementua du eta gero r azpizerrendako elementu kopurua zenbatu beharko da.

- **bikop**: Int motako zerrenda bat emanda, zerrendan zenbat elementu bikoiti dauden itzuliko du.

Adibideak:

```
bikop [] = 0
bikop [5, 8, 7, 8] = 2
bikop [7, 11, 9] = 0
bikop [0, 3, 3, 5] = 1
```

Eragiketaren **mota**:

```
bikop:: [Int] -> Int
```

Eragiketa definitzen duten **ekuazioak**:

```
bikop [] = 0                                    (1)
bikop (x:r)
        | bikoitia x    = 1 + (bikop r)         (2)
        | bakoitia x    = bikop r               (3)
```

Emandako zerrenda hutsa baldin bada (hau da, [] erakoa baldin bada), 0 elementu bikoiti daude zerrenda horretan. Emandako zerrenda hutsa ez bada, hau da, adibidez x:r erakoa baldin bada (zerrenda horretako lehenengo elementua x dela eta zerrendako beste elementu denez osatutako azpizerrenda r dela kontsideratuz), zerrendako lehenengo elementua bikoitia baldin bada (hau da, x bikoitia baldin bada) bikoiti kopurua 1 gehi "r azpizerrendako bikoiti kopurua" izango da. Baina zerrendako lehenengo elementua bakoitia baldin bada (hau da, x bakoitia baldin bada) bikoiti kopurua "r azpizerrendako bikoiti kopurua" izango da, x ez da zenbatu behar, ez baita bikoitia.

bikop funtzioaren definizioan lehendik definituta ditugun *bikoitia* eta *bakoitia* funtzioak erabili dira. Funtzio berriak definitzean lehendik definituta dauden funtzioak erabil daitezke laguntzaile moduan.

- **tartekatu**: t motako bi zerrenda emanda, bi zerrenda horietako elementuak tartekatuz lortzen den zerrenda itzuliko du funtzio honek. Lehenengo zerrendako lehenengo elementua zerrenda berriko lehenengo elementua izango da, bigarren zerrendako lehenengo elementua zerrenda berriko bigarren elementua izango da eta abar. Hasierako zerrenda biak luzera berekoak ez badira, errorea gertatuko da (errore-mezua aurkeztuko da).

  Adibidea:
  tartekatu [**7, 5, 4**] [8, 2, 0] = [**7**, 8, **5**, 2, **4**, 0]

  Eragiketaren **mota**:

```
tartekatu:: [t] -> [t] -> [t]
```

  Eragiketa definitzen duten **ekuazioak**:

```
tartekatu [] s
   | (luzera s) /= 0      = error "Luzera desberdineko zerrendak."      (1)
   | otherwise            = []                                          (2)

tartekatu (x:r) s
   | (luzera (x:r)) /= (luzera s) = error "Luzera desberdineko zerrendak."  (3)
   | otherwise                    = x:((leh s): (tartekatu r (hond s)))     (4)
```

  Kasu honetan 4 ekuazio eman dira *tartekatu* funtzioa definitzeko.

  Lehenengo bi ekuazioetan lehenengo zerrenda hutsa denean ([] erakoa denean) zer egin behar den zehazten da. Bigarren zerrenda (s zerrenda) hutsa ez bada, zerrenda bien luzera desberdina izango denez, errore-mezua aurkeztuko da eta

bestela, hau da, bigarren zerrendaren luzera [] zerrendaren luzeraren berdina denean, zerrenda biak hutsak direnez emaitzatzat ere zerrenda hutsa itzuliko da.

Beste bi ekuazioetan, lehenengo zerrenda hutsa ez denean (adibidez x:r erakoa denean, zerrendako lehenengo elementua x dela eta zerrendako gainontzeko elementuez osatutako azpizerrenda r dela kontsideratuz) zer egin behar den zehaztu da. Hor, (3) ekuazioan, x:r zerrendaren luzera eta s zerrendaren luzera desberdinak baldin badira, errore-mezua aurkeztuko dela adierazi da. Laugarren ekuazioan, x:r eta s zerrendek luzera bera dutenean bi zerrenda horietako elementuak tartekatuz osatutako zerrenda nola eraikiko den zehaztu da. Zerrenda berriko lehenengo elementua x:r zerrendako lehenengo elementua izango da (x elementua). Zerrenda berriko bigarren elementua s zerrendako lehenengo elementua izango da, baina elementu horrek izen berezirik ez duenez, *leh(s)* bezala adierazi behar da. Zerrenda berriaren gainontzeko zatia kalkulatzeko, lehenengo zerrendako (x:r zerrendako) gainontzeko elementuak, hau da, r azpizerrendako elementuak eta bigarren zerrendako (s zerrendako) gainontzeko elementuak tartekatu beharko dira. Baina s zerrendako gainontzeko elementuez osatutako azpizerrendari izenik ez diogunez eman, *hond(s)* moduan adierazi beharko da azpizerrenda hori.

- **tartekatu2**: t motako bi zerrenda emanda, bi zerrenda horietako elementuak tartekatuz lortzen den zerrenda itzuliko du baina kasu honetan lehenengo zerrendako lehenengo elementua zerrenda berriko bigarren elementua izango da, bigarren zerrendako lehenengo elementua zerrenda berriko lehenengo elementua izango da eta abar. Hasierako zerrenda biak luzera berekoak ez badira, errorea gertatuko da (errore-mezua aurkeztuko da).

  Adibidea:

  tartekatu2 [**7, 5, 4**] [8, 2, 0] = [8, **7**, 2, **5**, 0, **4**]

  Eragiketaren **mota**:
  tartekatu2:: [t] -> [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:

```
tartekatu2 [] s
  | (luzera s) /= 0      = error "Luzera desberdineko zerrendak."   (1)
  | otherwise            = []                                       (2)

tartekatu2 (x:r) s
  | luzera (x:r) /= (luzera s)   = error "Luzera desberdineko zerrendak."   (3)
  | otherwise                    = (leh s):(x: (tartekatu2 r (hond s)))     (4)
```

  Ekuazio hauek ematean, aurreko adibideko, hau da, *tartekatu* izeneko funtzioaren kasuko ideia bera jarraitu da, baina (4) ekuazioan zerrenda berria eraikitzean, elementuak kontrako ordenean ipini dira lortu nahi den zerrenda lortu ahal izateko. Hori da hain zuzen ere *tartekatu* eta *tartekatu2* izeneko funtzioen arteko desberdintasuna.

- **pos_bik_kendu**: t motako zerrenda bat emanda, posizio bikoitietako elementuak kenduz lortzen den zerrenda itzuliko du funtzio honek. Datu bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko du.

  pos_bik_kendu [7, **5**, 6, **8**, 2] = [7, 6, 2]

  Eragiketaren **mota**:
  pos_bik_kendu:: [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:
  pos_bik_kendu [] = []
  pos_bik_kendu (x:s)
  | hutsa_da s = x:[]
  | otherwise = x:(pos_bik_kendu (hond s))

- **pos_bak_kendu**: t motako zerrenda bat emanda, posizio bakoitietako elementuak kenduz lortzen den zerrenda itzuliko du funtzio honek. Datu bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko du.

  pos_bik_kendu [**7**, 5, **6**, 8, **2**] = [5, 8]

  Eragiketaren **mota**:
  pos_bak_kendu:: [t] -> [t]

  Eragiketa definitzen duten **ekuazioak**:
  pos_bak_kendu [] = []
  pos_bak_kendu (x:s)
  | hutsa_da s = []
  | otherwise = (leh s):(pos_bak_kendu (hond s))

7.4.4. Zerrendentzako eragile erlazionalak

Oinarrizko datu-moten atalean aipatutako eragile erlazionalak zerrendekin ere erabil daitezke.

Zerrenda bat beste bat baino txikiagoa edo handiagoa al den erabakitzeko hitzak alfabetikoki ordenatzeko erabiltzen den teknika bera erabiltzen da.

Esate baterako [3, 3, 4] zerrenda [5, 1] zerrenda baino txikiagoa da 3 zenbakia 5 baino txikiagoa delako.

7.4.5. String datu-mota: Char motako zerrendak

Aurredefinitutako String mota [Char] motaren baliokidea edo sinonimoa da. Beraz, String motako elementuak Char motako osagaiak dituzten zerrendak dira. Char motako elementuak 'a', '?' edo '7' eran idazten dira. Char motako zerrenda bat edo String motako elementu bat hiru eratara idatz daiteke:

'A':'z':'a':'r':'o':'a':[]

['A', 'z', 'a', 'r', 'o', 'a']

edo

"Azaroa"

Erosotasuna dela-eta, hirugarren formatua da ohikoena. Gainerakoan, Char motako zerrendak beste motetako zerrendak bezala erabil daitezke.

7.5. Errekurtsibitate gurutzatua

Bi funtziok elkarri deitzen diotenean errekurtsibitate gurutzatua dutela esaten da:

- **bikoitiag eta bakoitiag**: Errekurtsibitate gurutzatuaren bidez, zero baino handiagoa edo berdina den zenbaki bat bikoitia ala bakoitia den erabakiko duten funtzioak.

```
bikoitiag 7 = False
bikoitiag 12 = True
bakoitiag 7 = True
bakoitiag 12 = False
```

Funtzio bakoitzari dagozkion **mota** eta ekuazioak honako hauek dira:

```
bikoitiag:: Int -> Bool

bikoitiag x
  | x < 0          = error "Balio negatiboa."
  | x == 0         = True
  | otherwise      = bakoitiag (x – 1)

bakoitiag:: Int -> Bool

bakoitiag x
  | x < 0          = error "Balio negatiboa."
  | x == 0         = False
  | otherwise      = bikoitiag (x – 1)
```

Funtzio hauek ez dira batere eraginkorrak zenbaki bat bikoitia ala bakoitia den erabakitzeko, baina errekurtsibitate gurutzatuaren adibide gisa balio dute.

Adibideak:

5 zenbakia bikoitia al den erabakitzeko, honako kalkulu hauek burutuko lirateke:

```
bikoitiag 5 =
bakoitiag 4 =
bikoitiag 3 =
bakoitiag 2 =
bikoitiag 1 =
bakoitiag 0 = False
```

4 zenbakia bikoitia al den erabakitzeko, honako kalkulu hauek burutuko lirateke:

```
bikoitiag 4 =
bakoitiag 3 =
bikoitiag 2 =
```

```
bakoitiag 1 =
bikoitiag 0 = True
```

- **pos_bik_kendug eta pos_bak_kendug**: Errekurtsibitate gurutzatuaren bidez, datu gisa emandako zerrendako posizio bikoitietako eta posizio bakoitietako elementuak kenduko dituzten funtzioak.

```
pos_bik_kendug [7, 5, 6, 8, 2] = [7, 6, 2]
pos_bak_kendug [7, 5, 6, 8, 2] = [5, 8]
```

Jarraian funtzio bakoitzari dagozkion **mota** eta ekuazioak emango dira:

```
pos_bik_kendug:: [t] -> [t]

pos_bik_kendug [] = []
pos_bik_kendug (x:s) = x:(pos_bak_kendug s)

pos_bak_kendug:: [t] -> [t]

pos_bak_kendug [] = []
pos_bak_kendug (x:s) = pos_bik_kendug s
```

Adibideak:

[7, 5, 6, 8, 2] zerrendako posizio bikoitietan dauden elementuak kentzeko, honako kalkulu hauek burutuko lirateke:

```
pos_bik_kendug [7, 5, 6, 8, 2] = pos_bik_kendug (7:5:6:8:2:[]) =
7: (pos_bak_kendug (5:6:8:2:[])) =
7: (pos_bik_kendug (6:8:2:[])) =
7:6: (pos_bak_kendug (8:2:[])) =
7:6: (pos_bik_kendug (2:[])) =
7:6:2: (pos_bak_kendug []) = 7:6:2:[] = [7, 6, 2]
```

[7, 5, 6, 8, 2] zerrendako posizio bakoitietan dauden elementuak kentzeko, honako kalkulu hauek burutuko lirateke:

```
pos_bak_kendug [7, 5, 6, 8, 2] = pos_bak_kendug (7:5:6:8:2:[]) =
pos_bik_kendug (5:6:8:2:[]) =
5: (pos_bak_kendug (6:8:2:[])) =
5: (pos_bik_kendug (8:2:[])) =
5:8: (pos_bak_kendug (2:[])) =
5:8: (pos_bik_kendug []) = 5:8:[] = [5, 8]
```

7.6. Modularitatea Haskell-en

7.6.1. Moduluen definizioa

Haskell lengoaian moduluak definitzea eta modulu batetik beste modulu batean definitutako funtzioak erabiltzea oso erraza da.

Modulu bakoitza era independentean erabil daiteke baina modulu bat beste modulu batetik inportatu daiteke, hau da, modulu batean beste moduluren batean definitutako funtzioak erabil daitezke import aukerarekin.

Modulu bakoitza fitxategi desberdin batean joango da. Fitxategi bateko funtzioek modulu bat osa dezaten, fitxategiaren hasieran honako hau ipini beharko da:

module *Moduluaren_izena* **where**

Beste modulu batean definitu diren funtzioak erabili nahi badira, honako hau idatzi beharko da:

import *Beste_moduluaren_izena*

Modulu bat definitutakoan, modulu horretako funtzio denak esportagarriak dira. Bakarrik moduluko funtzio batzuk esportatu nahi badira, modulua honela definitu beharko da:

module *Moduluaren_izena* (*funtzioa_1*, *funtzioa_2*, …, *funtzioa_n*) **where**

Horrela, parentesien artean agertzen diren funtzioak bakarrik esportatuko dira.

Inportatzerakoan ere posible da beste modulu bateko funtzio batzuk bakarrik (ez guztiak) inportatzea. Horretarako, zein funtzio inportatu nahi diren zehaztu beharko da parentesien artean:

import *Beste_noduluaren_izena* (*funtzioa_1*, *funtzioa_2*, …, *funtzioa_n*)

Horrela, parentesien artean agertzen diren funtzioak bakarrik inportatuko dira.

Gainera, inportatzerakoan posible da funtzio batzuk ez eta beste denak inportatzea. Horretarako, zein funtzio ez diren inportatu behar zehaztu beharko da parentesien artean:

import *Beste_noduluaren_izena* **hiding** (*funtzioa_1*, *funtzioa_2*, …, *funtzioa_n*)

Horrela, parentesien artean agertzen ez diren funtzioak bakarrik inportatuko dira.

Azkenik, gerta daiteke modulu desberdinetan definitutako bi funtziok izen bera izatea. Bi modulu horiek inportatzen badira, errorea sortuko da. Izan ere, izen bera duten

funtzioak erabiltzean interpretatzaileak ez du jakingo bi funtzio horietatik zein erabili behar duen. Arazo hori saihesteko, funtzio bat erabiltzean zein modulutakoa den zehaztu beharra ipini dezakegu. Horretarako, moduluak inportatzean **qualified** hitza gehitu beharko da:

import qualified *Beste_noduluaren_izena*

Horrela importatuz gero, *Beste_noduluaren_izena* modulukoa den f funtzio bat erabiltzeko moduluaren izena, puntua, funtzioaren izena eta funtzioari dagozkion argumentuak idatzi beharko dira:

*Beste_noduluaren_izena.*f …

7.6.2. Aurreko ataletan definitutako funtzioak dituzten lau moduluren definizioa

Gai honetan orain arte definitu ditugun funtzioak kontuan hartuz, lau modulu definituko ditugu:

a) *Ez_errek* izeneko modulua oinarrizko motentzat definitutako funtzio ez errekurtsiboekin (pi2, f, bikoitia, bakoitia, hand3).

b) *Errek_zenb* izeneko modulua Int motarentzat definitutako funtzio errekurtsiboekin (bider, bneg, errusiar, zatios, zatihond).

c) *Zerrendak* izeneko modulua zerrendentzat definitutako funtzio errekurtsibo eta ez-errekurtsiboekin (leh, hond, hutsa_da, badago, luzera, bikop, tartekatu, tartekatu2, pos_bik_kendu, pos_bak_kendu).

d) *Errek_g* izeneko modulua errekurtsibitate gurutzatua duten funtzioekin (bikoitiag, bakoitiag, pos_bik_kendug, pos_bak_kendug).

Orain, lau modulu horiek erakutsiko dira:

a) "Ez_errek.hs" fitxategia

```
module Ez_errek where

-----------------------------------------------------------------

-- "pi2" funtzioa: Sarrerako daturik hartu gabe beti 3.1415 balioa
-- itzuliko duen funtzioa.

pi2:: Float
pi2 = 3.1415

-----------------------------------------------------------------

-- "f" funtzioa: Sarrerako datu bezala zenbaki oso bat hartu eta beti
-- 100 balioa itzuliko duen funtzioa.

f:: Int -> Int
f x = 100

-----------------------------------------------------------------

-- "bikoitia" funtzioa: Zenbaki oso bat emanda, zenbakia bikoitia
-- al den ala ez erabakiko duen funtzioa.

bikoitia:: Int -> Bool
bikoitia x = (x `mod` 2) == 0

-----------------------------------------------------------------

-- "bakoitia" funtzioa: Zenbaki oso bat emanda, zenbakia bakoitia
-- al den ala ez erabakiko duen funtzioa.

bakoitia:: Int -> Bool
bakoitia x = not (bikoitia x)

-----------------------------------------------------------------

-- "hand3" funtzioa: Hiru zenbaki oso emanda, balio handiena itzuliko
-- duen funtzioa.

hand3:: Int -> Int -> Int -> Int
hand3 x y z
        | x >= y && x >= z          = x
        | y > x && y >= z           = y
        | otherwise                 = z

-----------------------------------------------------------------
```

b) "Errek_zenb.hs" fitxategia

```
module Errek_zenb where


-- Oinarrizko errekurtsibitatea zenbakiekin


-----------------------------------------------------------------

import Ez_errek

{- Ez_errek.hs moduluan dagoen "bikoitia" funtzioa erabili nahi delako
   "errusiar" izeneko funtzioa definitzeko.
-}

-----------------------------------------------------------------

-- "bider" funtzioa: x balioa y aldiz batuz x * y kalkulatuko duen
-- funtzioa. Kasu berezi bezala, y balioa negatiboa baldin bada,
-- errore-mezua aurkeztuko du.

bider:: Int -> Int -> Int
bider x y
   | y < 0                 = error "Bigarren balioa negatiboa da."
   | x == 0 || y == 0      = 0
   | x == 1                = y
   | otherwise             = x + bider x (y - 1)

-----------------------------------------------------------------

-- "bneg" funtzioa: x * y balioa batuketaren bidez kalkulatuko duen
-- funtzioa. y balioa negatiboa denean ere kalkulua ondo egingo da.
-- Aurretik definitu den "bider" funtzioa erabiliko da laguntzaile
-- bezala. "bneg" funtzioa berez ez da errekurtsiboa, ez baitio
-- bere buruari deitzen. Errekurtsibitatea "bider" funtzioan dago.

bneg:: Int -> Int -> Int
bneg x y
   | x == 0 || y == 0      = 0
   | (x < 0) && (y < 0)            = bider (-x) (-y)
   | (x > 0) && (y > 0)            = bider x y
   | (x < 0) && (y > 0)            = bider x y
   | (x > 0) && (y < 0)            = bider (-x) (-y)

-----------------------------------------------------------------

-- "errusiar" funtzioa: Errusiar biderketa bezala ezagutzen den
-- metodoari jarraituz x * y kalkulatuko duen funtzioa.
-- Kasu berezi bezala, y balioa negatiboa baldin bada, errore-mezua
-- aurkeztuko du.

errusiar:: Int -> Int -> Int
errusiar x y
   | y < 0                 = error "Bigarren balioa negatiboa da."
   | y == 0                = 0
   | bikoitia y            = errusiar (x + x) (y `div` 2)
   | otherwise             = x + errusiar (x + x) (y `div` 2)
```

```
-----------------------------------------------------------------

-- "zatios" funtzioa: y balioa x balioari zenbat aldiz kendu dakiokeen
-- zenbatuz, x eta y-ren arteko zatidura osoa kalkulatuko duen funtzioa.
-- Kasu berezi bezala, zatitzailea zero denean, errore-mezua aurkeztuko du.
-- Gainera, x edo y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

zatios:: Int -> Int -> Int
zatios x y
  | y == 0                = error "Zatitzailea 0."
  | (x < 0) || (y < 0)    = error "Gutxienez bietako bat negatiboa da."
  | x < y          = 0
  | otherwise             = 1 + zatios (x - y) y

-----------------------------------------------------------------

-- "zatihond" funtzioa: y balioa baino txikiagoa den balio bat eduki
-- arte, x balioari eta ondoren lortuko diren kendurei y balioa kenduz,
-- x eta y-ren arteko zatidura osoaren hondarra kalkulatuko duen funtzioa.
-- Kasu berezi bezala, zatitzailea zero denean, errore-mezua aurkeztuko du.
-- Gainera, x edo y negatiboa baldin bada ere, errore-mezua aurkeztuko du.

zatihond:: Int -> Int -> Int
zatihond x y
  | y == 0                = error "Zatitzailea 0."
  | (x < 0) || (y < 0)    = error "Gutxienez bietako bat negatiboa da."
  | x < y          = x
  | otherwise             = zatihond (x - y) y

-----------------------------------------------------------------
```

import Ez_errek ipiniz, modulu honetatik *Ez_errek* moduluko funtzioak erabilgarri daude.

c) "Zerrendak.hs" fitxategia

```
module Zerrendak where

-----------------------------------------------------------------

import Ez_errek

{- Ez_errek.hs moduluan dauden "bikoitia" eta "bakoitia" funtzioak
   erabili nahi direlako "bikop" izeneko funtzioa definitzean.
-}

-----------------------------------------------------------------

-- "leh" funtzioa: Zerrenda bat emanda, zerrendako lehenengo
-- elementua itzuliko duen funtzioa.
-- Zerrenda hutsa baldin bada, errore-mezua itzuliko du.

leh:: [t] -> t
leh [] = error "Zerrenda hutsa. Ez dago lehenengo elementurik."
leh (x:s) = x

-----------------------------------------------------------------
```

```
-- "hond" funtzioa: Zerrenda bat emanda, zerrendako lehenengo
-- elementua kenduz lortuko den zerrenda itzuliko duen funtzioa.
-- Zerrenda hutsa baldin bada, errore-mezua itzuliko du.

hond:: [t] -> [t]
hond [] = error "Zerrenda hutsa. Ez dago hondarrik."
hond (x:s) = s

-----------------------------------------------------------------

-- "hutsa_da2 funtzioa: Zerrenda bat emanda, zerrenda hutsa al den
-- ala ez erabakiko duen funtzioa.

hutsa_da:: [t] -> Bool
hutsa_da [] = True
hutsa_da (x:s) = False

-----------------------------------------------------------------

-- "badago" funtzioa: Elementu bat eta zerrenda bat emanda,
-- elementua zerrendan agertzen al den ala ez erabakiko duen funtzioa.

badago:: t -> [t] -> Bool
badago x [] = False
badago x (y:s)
        | x == y = True
        | x /= y  = badago x s

-----------------------------------------------------------------

-- "luzera" funtzioa: Zerrenda bat emanda, zerrendako
-- elementu-kopurua kalkulatuko duen funtzioa.

luzera:: [t] -> Int
luzera [] = 0
luzera (x:r) = 1 + luzera r

-----------------------------------------------------------------

-- "bikop" funtzioa: Zerrenda bat emanda, zerrendako elementu
-- bikoitien kopurua kalkulatuko duen funtzioa.

bikop:: [Int] -> Int
bikop [] = 0
bikop (x:r)
        | bikoitia(x)      = 1 + bikop r
        | bakoitia(x)      = bikop r

-----------------------------------------------------------------

-- "tartekatu" funtzioa: Bi zerrenda emanda, zerrenda bietako
-- elementuak tartekatuz lortuko den zerrenda kalkulatuko duen funtzioa.
-- Hasteko lehenengo zerrendatik hartu behar da.
-- Zerrenda biek luzera bera ez badute, errore-mezua aurkeztuko du.

tartekatu:: [t] -> [t] -> [t]
tartekatu [] s
        | luzera s /= 0    = error "Luzera desberdineko zerrendak."
        | otherwise                = []
tartekatu (x:r) s
```

```
        | luzera (x:r) /= luzera s  = error "Luzera desberdineko zerrendak."
        | otherwise                = x:((leh s): (tartekatu r (hond s)))

-----------------------------------------------------------------

-- "tartekatu2" funtzioa: Bi zerrenda emanda, zerrenda bietako
-- elementuak tartekatuz lortuko den zerrenda kalkulatuko duen funtzioa.
-- Hasteko bigarrengo zerrendatik hartu behar da.
-- Zerrenda biek luzera bera ez badute, errore-mezua aurkeztuko du.

tartekatu2:: [t] -> [t] -> [t]
tartekatu2 [] s
        | luzera s /= 0    = error "Luzera desberdineko zerrendak."
        | otherwise        = []
tartekatu2 (x:r) s
        | luzera (x:r) /= luzera s  = error "Luzera desberdineko zerrendak."
        | otherwise                 = (leh s):(x:(tartekatu2 r (hond s)))

-----------------------------------------------------------------

-- "pos_bik_kendu" funtzioa: Zerrenda bat emanda, posizio bikoitietako
-- elementuak kenduz lortuko den zerrenda itzuliko duen funtzioa.

pos_bik_kendu:: [t] -> [t]

pos_bik_kendu [] = []
pos_bik_kendu (x:s)
        | hutsa_da s       = x:[]
        | otherwise        = x:(pos_bik_kendu (hond s))

-----------------------------------------------------------------

-- "pos_bak_kendu" funtzioa: Zerrenda bat emanda, posizio bakoitietako
-- elementuak kenduz lortuko den zerrenda itzuliko duen funtzioa.

pos_bak_kendu:: [t] -> [t]

pos_bak_kendu [] = []
pos_bak_kendu (x:s)
        | hutsa_da s       = []
        | otherwise        = (leh s):(pos_bak_kendu (hond s))

-----------------------------------------------------------------
```

[] eta (:) eragile eraikitzaileak eta (++) eragilea aurredefinituta daude Haskell-en.

import Ez_errek ipiniz, modulu honetatik *Ez_errek* moduluko funtzioak erabilgarri daude.

d) "Errek_g.hs" fitxategia

```
module Errek_g where

-----------------------------------------------------------------

-- Errekurtsibitate gurutzatua

-----------------------------------------------------------------
-----------------------------------------------------------------

-- "bikoitiag" eta "bakoitiag" funtzioak: Errekurtsibitate
-- gurutzatuaren bidez, 0 baino handiagoa edo berdina den zenbaki
-- oso bat bikoitia ala bakoitia den erabakiko duten funtzioak.

bikoitiag:: Int -> Bool

bikoitiag x
  | x < 0              = error "Balio negatiboa."
  | x == 0             = True
  | otherwise          = bakoitiag (x - 1)

---
---

bakoitiag:: Int -> Bool

bakoitiag x
  | x < 0              = error "Balio negatiboa."
  | x == 0             = False
  | otherwise          = bikoitiag (x - 1)

-----------------------------------------------------------------
-----------------------------------------------------------------

-- "pos_bik_kendug" eta "pos_bak_kendug" funtzioak: Errekurtsibitate
-- gurutzatuaren bidez, posizio bikoitietako eta posizio bakoitietako
-- elementuak kentzen dituzten funtzioak.

pos_bik_kendug:: [t] -> [t]

pos_bik_kendug [] = []
pos_bik_kendug (x:s) = x:(pos_bak_kendug s)

---
---

pos_bak_kendug:: [t] -> [t]
```

```
pos_bak_kendug [] = []
pos_bak_kendug (x:s) = pos_bik_kendug s

----------------------------------------------------------------
----------------------------------------------------------------
```

7.6.3. Aurredefinitutako moduluak

Haskell lengoaian badaude aurredefinitutako hainbat modulu. Modulu horietako bakoitzean funtzio batzuk defituta daude, aurredefinitutako funtzioak. Hemen modulu horietako batzuk bakarrik aipatuko dira:

a) *Prelude*: oinarrizko modulua da. Modulu honetako funtzioak beti erabilgarri daude, ez dago modulu hau inportatu beharrik. Modulu honetan Haskell-eko hainbat motentzat oinarrizko funtzioak daude (zenbakientzat, Char motarentzat, zerrendentzat, tuplentzat eta abar).

b) *Data.Char*: Char motaren gaineko funtzio batzuk ditu modulu honek. Hurrengo atalean azalduko dira modulu honetako funtzio garrantzitsuenak.

c) *Data.List*: zerrendekin zerikusia duten funtzio batzuk daude definituta modulu honetan. Aurrerago azalduko dira funtzio horietako batzuk.

d) *System.IO*: sarrera/irteerarekin eta fitxategiekin zerikusia duten funtzio batzuk daude modulu honetan. Sarrera/irteerari eta fitxategiei dagozkien ataletan modulu honetako funtzio batzuk azalduko dira.

7.6.4. Data.Char modulua

Data.Char moduluan Char motarentzat definitutako funtzioa batzuk daude. Garrantzitsuenak honako hauek dira:

a) *toLower*: maiuskula bat manda, dagokion minuskula itzuliko du. Maiuskula bat ez den karaktere bat ematen bazaio, karaktere bera itzuliko du.

b) *toUpper*: minuskula bat manda, dagokion maiuskula itzuliko du. Minuskula bat ez den karaktere bat ematen bazaio, karaktere bera itzuliko du

c) *isAlpha*: karaktere bat emanda, True itzuliko du karakterea letra bat baldin bada eta False itzuliko du kontrako kasuan.

d) *isDigit*: karaktere bat emanda, True itzuliko du karakterea digitu bat baldin bada eta False itzuliko du kontrako kasuan.

e) *isAscii*: karaktere bat emanda, True itzuliko du karakterea ASCII karaktere bat baldin bada eta False itzuliko du kontrako kasuan. ASCII karaktereak '\0'-tik '\127'-ra doazenak dira. Tarte horretan digituak ('\48', '\49', …), maiuskulak

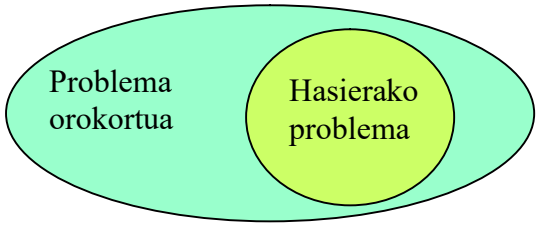
('\65', '\66', …), minuskulak ('\97', '\98', …) eta gainerako ohiko karaktereak daude (adibidez, '\93' espresioa ']' karaktereari dagokiona da).

7.7. Murgilketa

Atal honetan *murgilketa* izenarekin ezagutzen den teknika azalduko da. Ebatzi beharreko problema zuzenean ebatzi beharrean, problema hori orokortu eta orokortze horren bidez lortutako problema berria ebaztean datza teknika hau. Orokortuz lortutako problema berria ebatzitakoan, hasierako problema ere ebatzita geratuko da, hau da, problema orokortua ebatziz hasierako problema ere ebatzita geratuko da, hasierako problema problema orokortuaren kasu partikular bat izango baita. Horregatik, hasierako problema problema orokortuan **murgilduta** geratu dela esan daiteke.

Problema bat orokortzeko, hasierako problemari parametro berriak gehituko zaizkio murgilketaren teknikan. Parametro berri horiek helburu desberdinak izan ditzakete:

- Tarte bat zeharkatzeko indize gisa erabiliko diren parametroak izan daitezke.
- Bukaerako emaitza kalkulatzeko erabiliko diren behin-behineko emaitzak edo emaitza partzialak gordetzeko erabiliko diren parametro berriak ere izan daitezke (zenbatzaile moduan edo baturak gordetzeko edo zerrendak gordetzeko, eta abar).



Problema batzuetan, soluzio errekurtsiboa lortzea nahi baldin bada, murgilketaren teknika erabiltzea derrigorrezkoa da. Beste problema batzuetan murgilketa ez da beharrezkoa baina hala ere erabil daiteke; izan ere, gehienetan eraginkortasun-maila hobetzen da murgilketa erabiliz.

7.7.1. Murgilketarik gabeko soluzio errekurtsiboak

7.3.8, 7.4.3 eta 7.5 ataletan aurkeztu diren funtzio errekurtsiboetan murgilketa ez da erabili. Funtzio horiek 7.6 atalean definitutako *Errek_zenb*, *Zerrendak* eta *Errek_g* moduluetan jaso dira. Beraz funtzio horiek murgilketarik gabeko soluzio errekurtsiboen adibidetzat har ditzakegu. Kasu horietan hasierako problema zuzenean ebatzi da, orokortu gabe.

7.7.2. Tarteak zeharkatzeko indize moduan erabiliko diren parametroak gehitzean oinarritzen den murgilketa

Era honetako murgilketetan, tarte bat zeharkatzeko indize gisa erabiliko den parametro berri bat edukiko dugu.

a) Osoa den zenbaki positibo baten zatitzaileen zerrenda: *zatizer*

Osoa den zenbaki positibo baten zatitzaileen zerrenda kalkulatzeko, x zatitzen duten [1..x] tarteko zenbakiak aurkitu behar dira.

zatizer izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, x zenbaki oso bat emanda, [1..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer:: Int -> [Int]
zatizer x
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | otherwise             = ???
```

Baina ezin dugu zuzenean x balioan oinarritutako soluzio errekurtsiborik planteatu. Faktorialaren definizioa hartuz, x zenbaki baten faktoriala x – 1 zenbakiaren faktoriala erabiliz kalkula daiteke. Zerrenden gainean definitutako funtzio errekurtsiboak aztertuz, adibidez zerrenda baten luzera, elementu bat gutxiago duen azpirrendaren luzera erabiliz kalkula daiteke. Baina zenbaki baten zatitzaileen zerrendaren kasuan, x zenbaki baten zatitzaileen zerrendak ez du zerikusirik x – 1 zenbakiaren zatitzaileen zerrendarekin. Adibide gisa, 8 eta 7 zenbakien zatitzaileen zerrendak har ditzakegu. Alde batetik, 8 zenbakiaren zatitzaileen zerrenda [1, 2, 4, 8] da. Beste aldetik, 7 zenbakiaren zatitzaileen zerrenda [1, 7] da. Beraz, ezinezkoa da 8ren zatitzaileen zerrenda 7ren zatitzaileen zerrendatik kalkulatzea.

Zatitzaileen zerrenda kalkulatzean, errekurtsibitatea ez dago x zenbakian, errekurtsibitatea [1..x] tartean dago. Horrela, x zenbakiaren zatitzaileen zerrenda kalkulatzeko, tarte horretako lehenengo elementuak x zatitzen al duen ala ez aztertu behar da (beraz hasieran 1 zenbakiak x zenbakia zatitzen al duen aztertuko da). Gero, x zenbakiaren beste zatitzaileak aurkitzeko, [2..x] tarteko zenbakiak aztertu beharko dira. [2..x] tartea finkatu ondoren, [2..x] tartekoak diren x zenbakiaren zatitzaileen zerrenda, 2 zenbakiak x zenbakia zatitzen al duen aztertuz eta [3..x] tartean x zenbakiaren zatitzaile gehiago bilatuz lortuko da. Garapen honetan nabaria den bezala, errekurtsibitatea tarte horretan dago eta tarte hori gero eta txikiagoa da. Orokorrean, tarteko beheko muga bm baldin bada, [bm..x] tartekoak diren x zenbakiaren zatitzaileak, bm zenbakiak x zatitzen al duen aztertuz eta gero [bm + 1..x] tartean x-ren zatitzaileak bilatuz lortuko dira. Eraginkortasuna dela eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [1..x div 2] tartera muga daiteke.

Beraz, murgilketa x zenbakiaren zatitzaileak [1..x] tartean bilatu beharrean [bm..x] tarte orokor batean bilatzean oinarrituko da. Definituko dugun funtzio berrian bi parametro edukiko ditugu, x eta bm. Hor bm balioak x zenbakiaren zatitzaileak bilatzerakoan kontsideratu beharreko tartearen beheko muga zein den adierazteko balioko digu.

Funtzio berriari zatizer_lag deituko diogu, zatizer funtzioa definitzeko laguntzaile gisa erabiliko baitugu.

```
-- Funtzio honek, x eta bm bi zenbaki oso emanda, [bm..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer_lag:: Int -> Int -> [Int]
zatizer_lag x bm
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | bm <= 0               = error "Beheko muga ez da positiboa."
        | bm > x                = []
        | bm > (x `div` 2)      = [x]
        | x `mod` bm == 0       = bm : (zatizer_lag x (bm + 1))
        | otherwise             = zatizer_lag x (bm + 1)
```

zatizer_lag funtzioa definitu ondoren, zatizer funtzioaren definizioa eman dezakegu, izan ere zatizer funtzioa zatizer_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 1 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda, [1..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu
-- murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer:: Int -> [Int]
zatizer x = zatizer_lag x 1
```

Definizioan ikus daiteke zatizer funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina zatizer_lag funtzioa errekurtsiboa da.

zatizer_lag funtzioak ebazten duen problema zatizer funtzioak ebazten duena baino orokorragoa da; izan ere, zatizer_lag funtzioaren bidez esate baterako, 5 baino handiagoak edo berdinak diren 28 zenbakiaren zatitzaileak kalkula ditzakegu:

```
zatizer_lag 28 5
```

erantzuna [7, 14, 28] izango litzateke.

zatizer funtzioarekin aldiz, zatitzaile denak kalkulatuko dira beti, ez dago beheko muga aukeratzerik:

```
zatizer 28
```

erantzuna [1, 2, 4, 7, 14, 28] izango litzateke.

zatizer_lag erabiliz 28ren zatitzaile denen zerrenda kalkulatzea nahi izanez gero, honako hau ipini beharko litzateke:

```
zatizer_lag 28 1
```

b) 1 edo handiagoa den zenbaki bat lehena al den erabakitzen duen funtzioa:

1 edo handiagoa den zenbaki oso bat lehena izango da zehazki bi zatitzaile baldin baditu. Adibidez 2, 3, 5, 7, 11, 13 eta 17 zenbakiak lehenak dira zehazki bi zatitzaile dituztelako (1 eta zenbakia bera). Bestalde, 1 zenbakia ez da lehena zatitzaile bakarra duelako eta, esate baterako, 4, 6, 8, 9, 10, 12, 14 eta 16 ez dira lehenak bi zatitzaile baino gehiago dituztelako.

Zenbaki bat lehena al den erabakitzeko aukera bat zenbaki horren zatitzaileen zerrenda kalkulatu eta zerrenda horretan justu bi zenbaki al dauden ala ez aztertzea izango litzateke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du murgilketa erabili gabe.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena_gb:: Int -> Bool
lehena_gb x
        | x <= 0                    = error "Zenbakia ez da positiboa."
        | luzera (zatizer x) == 2   = True
        | luzera (zatizer x) /= 2   = False
```

Zenbaki bat lehena al den erabakitzen duen funtzio hori ez da errekurtsiboa, ez baitio bere buruari deitzen eta gainera murgilketa erabili gabe definitu da. Hala ere, gogoratu luzera eta zatizer funtzioak errekurtsibitatean oinarrituta daudela.

lehena_
gb funtzioaren arazoa eraginkortasun eza da. Izan ere, 28 lehena al den erabakitzeko, hasteko bere zatitzaileen zerrenda kalkulatuko du: [1, 2, 4, 7, 14, 28]. Zatitzaileen zerrenda kalkulatu ondoren, zatitzaileak zenbatuko ditu eta zehazki bi ez badira, False itzuliko du eta zehazki bi badira, True itzuliko du. Zenbaki bat lehena al den erabakitzeko era hau ez da eraginkorra, 2 zenbakiak 28 zenbakia zatitzen duela ikusi ondoren bai baitakigu 28 ez dela lehena, 1 eta 28 zenbakiez gain gutxienez hirugarren zatitzaile bat ere baduelako. Beraz, zatitzaile gehiago bilatzen jarduteak ez du zentzurik. Adibide bezala 1001 zenbakia hartzen badugu, gauza bera gertatzen da. Izan ere, lehena_gb funtzioari jarraituz, hasteko 1001en zatitzaileen zerrenda kalkulatuko litzateke: [1, 7, 11, 13, 77, 91, 143, 1001]. Gero, zerrenda horretako elementuak zenbatuko lirateke eta, zehazki bi ez daudenez, False itzuliko litzateke. Adibide honetan ere, 1001en zatitzaileak bilatzen hasitakoan 7 zenbakiak 1001 zatitzen duela ikustean badakigu 1001 ez dela lehena eta 1001en zatitzaile gehiago bilatzen jarraitzeak ez du zentzurik.

Orain zatizer funtzioa erabili gabe zuzenean x zenbakia lehena al den erabakitzen duen funtzio errekurtsibo bat definitzea da gure helburua. Plantamendu berri honetan, lehena ez den 1 zenbakia kasu berezi gisa hartuko da eta beste edozein x zenbakirentzat, x lehena izango da [2..x – 1] tartean x zenbakiaren zatitzailerik ez badago. Adibidez 7 zenbakiarentzat [2..6] tartean ez dago zatitzailerik eta ondorioz 7 lehena da. Baina 8 zenbakiaren kasuan, [2..7] tartean gutxienez zatitzaile bat badago (2 eta 4 zenbakiak 8ren zatitzaileak dira)

eta ondorioz 8 ez da lehena. Era berean, 28 zenbakiarentzat [2..27] tartean gutxienez zatitzaile bat badago (2, 4, 7 eta 14 zenbakiak 28ren zatitzaileak dira) eta ondorioz 28 ez da lehena. Bestalde, 2 zenbakiaren kasuan [2..2 – 1] tartea, hau da, [2..1] tartea hutsa da (goiko muga beheko muga baino txikiagoa delako) eta tarte hutsean 2ren zatitzailerik ezin denez egon, 2 zenbakia lehena da.

Orain badakigu zenbaki bat lehena ez dela erabakitzeko ez dagoela bere zatitzaile denak aurkitu beharrik, [2..x – 1] tartekoa den bere zatitzaile bat aurkitzearekin nahikoa da. Beraz, eraginkorragoa den funtzio bat lortzeko ideia hori hartuko dugu kontuan:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du [2..x – 1] tartean zatitzailerik ba al
-- duen aztertuz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena:: Int -> Bool
lehena x
        | x <= 0                    = error "Zenbakia ez da positiboa."
        | otherwise                 = ???
```

Baina x zenbakian oinarritutako soluzio errekurtsiborik ezin da aurkitu. Hemen ere aurretik definitu den zatizer funtzioarekin genuen arazo bera daukagu. Izan ere, x lehena izatea edo ez izatea ez dago x – 1 lehena izatearekin edo ez izatearekin erlazionatuta. Adibide gisa, alde batetik 3 lehena da eta 2 ere lehena da. Beste aldetik, 16 ez da lehena eta 15 ere ez da lehena. Baina 18 ez da lehena eta 17 bai.

Zenbaki baten zatitzaileak kalkulatzean bezala, kasu honetan ere errekurtsibitatea ez dago x zenbakian, errekurtsibitatea [2..x – 1] tartean dago. Beraz, x zenbakiak [2..x – 1] tartean zatitzailerik ba al duen erabakitzeko, tarte horretako lehenengo zenbakiak x zatitzen al duen aztertuko da (hasieran 2 zenbakiak x zatitzen al duen erabakiko da). Zatitzen badu, badakigu x ez dela lehena eta prozesua hor bukatuko da, ez dago zatitzaile gehiago bilatu beharrik. Ez badu zatitzen, [3..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen aztertzen jarraitu beharko da. [3..x – 1] tartea finkatu ondoren, tarte horretako lehenengo zenbakiak, hau da, 3 zenbakiak x zatitzen al duen begiratu beharko da. Zatitzen badu, badakigu x ez dela lehena eta prozesua hor bukatuko da, ez dago zatitzaile gehiago bilatu beharrik. Ez badu zatitzen, [4..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen aztertzen jarraitu beharko da. [4..x – 1] tartea finkatu ondoren, tarte horretako lehenengo zenbakiak, hau da, 4 zenbakiak x zatitzen al duen begiratu beharko da. Prozesuak x zenbakiaren zatitzaile bat aurkitu arte edo x zenbakiaren zatitzailerik ezingo dela aurkitu ziur egon arte jarraitu beharko du. Eraginkortasuna dela eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [2..x div 2] tartera muga daiteke.

Garapen honetan nabaria da errekurtsibitatea bilaketa-tarte horretan dagoela eta bilaketa-tarte hori gero eta txikiagoa dela. Orokorrean, bilaketa-tarteko beheko muga bm baldin bada, [bm..x – 1] tartekoa den x zenbakiaren zatitzailerik ba al dagoen jakiteko, bm zenbakiak x zatitzen al duen aztertu beharko da eta, bm ez

bada x zenbakiaren zatitzailea, [bm + 1..x – 1] tartean x zenbakiaren zatitzailerik ba al dagoen begiratzen jarraitu beharko da.

Beraz, murgilketa x zenbakiak [2..x – 1] tartean zatitzailerik ba al duen aztertzean oinarritu beharrean [bm..x – 1] tarte orokor batean zatitzailerik ba al duen aztertzean oinarrituko da. Definituko dugun funtzio berrian bi parametro edukiko ditugu, x eta bm. Hor, bm balioak x zenbakiak zatitzailerik ba al duen erabakitzean kontsideratu beharreko tartearen beheko muga zein den adierazteko balioko digu.

Funtzio berriari lehena_lag deituko diogu, lehena funtzioa definitzeko laguntzaile moduan erabiliko baitugu.

```
-- Funtzio honek, x eta bm bi zenbaki oso emanda, True itzuliko du
-- x zenbakiak [bm..x – 1] tartean zatitzailerik ez badu eta bestela
-- False itzuliko du.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 1 edo txikiagoa baldin bada, errorea.
-- Kasu berezi gisa, x zenbakiaren balioa 1 denean False itzuliko du.

lehena_lag:: Int -> Int -> Bool
lehena_lag x bm
        | x < 0 || x == 0       = error "Zenbakia ez da positiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 1                = False
        | bm > (x – 1)          = True
        | x `mod` bm == 0       = False
        | otherwise             = lehena_lag x (bm + 1)
```

Definizio horretan, laugarren baldintza hobetu daiteke bm > (x – 1) ipini beharrean bm > (x `div` 2) ipiniz. Orain, lehena_lag funtzioa definitu ondoren, lehena funtzioaren definizioa eman dezakegu. Izan ere, lehena funtzioa lehena_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 2 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- lehena al den erabakiko du [2..x - 1] tartean zatitzailerik ba al
-- duen aztertuz eta murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

lehena:: Int -> Bool
lehena x = lehena_lag x 2
```

Definizioan ikus daiteke lehena funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina lehena_lag funtzioa errekurtsiboa da.

lehena_lag funtzioak ebazten duen problema lehena funtzioak ebazten duena baino orokorragoa da. Izan ere, lehena_lag funtzioaren bidez esate baterako, 27 zenbakiak 10 edo handiagoa den zatitzailerik ba al duen jakin dezakegu. Baldintza hori betetzen duen zatitzailerik ez badago, True itzuliko du eta baldintza hori betetzen duen zatitzaileren bat baldin badago, orduan False itzuliko du:

lehena_lag 27 10

[10..26] tartean 27ren zatitzailerik ez dagoenez, erantzuna True izango litzateke.

lehena izeneko funtzioarekin aldiz 2 zenbakitik abiatuta aztertzen da zatitzailerik ba al dagoen ala ez:

lehena 27

[2..26] tartean 27ren zatitzaileak badaudenez, erantzuna False izango litzateke eta horrek 27 ez dela lehena esan nahi du.

lehena_lag funtzioa erabiliz 27 zenbakia [2..26] tartean zatitzailerik ez duen zenbaki bat al den erabaki nahi badugu, honako hau ipini beharko da:

lehena_lag 27 2

27 zenbakia [2..26] tartean zatitzailerik ez duen zenbaki bat ez denez, kasu honetan False balioa itzuliko luke funtzioak.

Bigarren parametroaren balioaren arabera, 27 zenbakiarentzat lehena_lag funtzioak True edo False itzuli dezakeela ikus dezakegu:

lehena_lag 27 10 → True
lehena_lag 27 2 → False
lehena_lag 27 5 → False

7.7.3. Behin-behineko emaitzak gordetzeko balio duten parametroak gehitzean oinarritutako murgilketa

Era honetako murgilketan, geroago edo bukaeran behin betiko emaitza kalkulatzeko erabiliko den emaitza partziala (edo behin-behineko emaitza) gordetzeko balio duen parametro berri bat izango du problema orokortuak.

a) Luzeren zerrenda: *azpiluz*

Zenbaki osozko zerrenda bat emanda, azpiluz izeneko funtzioak jarraian dauden balio berdinez osatutako azpizerrenden luzerez osatutako zerrenda kalkulatu behar du:

azpiluz [5, 5, 9, 8, 8, 8, 8, 7, 7, 8, 0, 0, 7] = [2, 1, 4, 2, 1, 2, 1]

azpiluz [3, 3, 3, 3, 3, 3, 3, 3, 3] = [9]

Sarrera gisa emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Sarrerako datua hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz:: [Int] -> [Int]

azpiluz [] = []
azpiluz (x:s)
        | hutsa_da s            = 1:[]
        | x == (leh s)          = ???
        | x /= (leh s)          = ???
```

Baina kontua da nola gorde azpizerrenda baten luzera. Luzera hori ez da behin betikoa azpizerrenda bukatu arte. Gainera azpizerrenda bakoitzaren behin betiko luzera bakarrik gorde nahi da.

Azpizerrenda bakoitzaren kasuan, azpizerrendaren behin-behineko luzera gordez joan beharko dugu behin betiko luzera ezagutu arte. Azpizerrendaren behin betiko luzera kalkulatutakoan, eraikitzen ari garen luzeren zerrendan gorde dezakegu luzera hori.

Murgilketarik gabe ere egin daiteke kalkulu hori, baina ez zuzenean, beste funtzio batzuk laguntzaile moduan definitu beharko dira. Hori da orain egingo duguna. Horretarako, berdinak_zenbatu eta berdinak_kendu funtzio laguntzaileak definituko dira eta, gero, bi funtzio laguntzaile horiek erabiliz baina murgilketa erabili gabe, elementu berdinez osatutako azpizerrenden luzerez osatutako zerrenda kalkulatuko duen azpiluz_gb funtzioaren definizioa emango da:

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrendaren luzera itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_zenbatu:: [Int] -> Int

berdinak_zenbatu [] = error "Zerrenda hutsa."

berdinak_zenbatu (x:s)
        | hutsa_da s            = 1
        | x == (leh s)          = 1 + (berdinak_zenbatu s)
        | x /= (leh s)          = 1
```

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda kenduz geratzen den zerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_kendu:: [Int] -> [Int]

berdinak_kendu [] = error "Zerrenda hutsa."

berdinak_kendu (x:s)
```

```
        | hutsa_da s          = []
        | x == (leh s)        = berdinak_kendu s
        | x /= (leh s)        = s
```

```
-- Funtzio honek, zenbaki osozko zerrenda
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Kalkulua murgilketarik gabe egiten du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz_gb:: [Int] -> [Int]

azpiluz_gb [] = []

azpiluz_gb (x:s) = (berdinak_zenbatu (x:s)):(azpiluz_gb (berdinak_kendu (x:s)))
```

azpiluz_gb funtzioa erabiliz, azpizerrenda bakoitza bi aldiz zeharkatu beharko da: lehenengo aldian elementuak zenbatuko dira eta bigarren aldian kendu egingo dira. Zerrenda luzeen kasuan azpizerrenda bakoitza bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin. Jarraian murgilketa erabiliz, azpizerrenda bakoitza behin zeharkatuko duen soluzio bat emango da.

Murgilketan oinarritutako soluzioan, azpizerrenda bakoitzaren behin-behineko luzera gordez joateko zenbatzaile gisa erabiliko den parametro berri bat behar da. Beraz, kasu honetan murgilketa aplikatzeko, parametro gisa zenbaki osozko zerrenda bat edukitzeaz gain beste zenbaki bat ere izango duen funtzio laguntzailea kontsideratuko da. Parametro berri hori azpizerrenda bakoitzaren luzera kalkulatzeko zenbatzaile moduan erabiliko da. Azpizerrenda bakoitzarekin hastean zenbatzaile hori 0 balioarekin hasieratu beharko da eta azpizerrenda bakoitza bukatzean, zenbatzaileak une horretan duen balioa eraikitzen ari garen luzeren zerrendan gorde beharko da.

Funtzio laguntzailea definitzean, zenbatzaile moduan erabiliko den parametroak ezin du 0 konstantea izan, funtzioak definitzean argumentuek parametro orokorrak izan behar baitute. Hori dela-eta, funtzio laguntzailearen esanahia honako hau izango da:

> Zenbaki osozko zerrenda bat eta zenbaki oso bat emanda, jarraian dauden balio berdinez osatutako azpizerrenden luzerez osatutako zerrenda kalkulatuko da. Baina lehenengo azpizerrendaren kasuan azpizerrenda horren luzera gehi bigarren parametroaren balioa gordeko da.

Funtzio berriari azpiluz_lag deituko diogu, azpiluz funtzioa definitzeko laguntzaile moduan erabiliko baitugu.

```
-- Funtzio honek, zenbaki osozko zerrenda bat eta zenbaki oso
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Lehenengo azpizerrendaren kasuan, bere luzera gehi luz itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpiluz_lag:: [Int] -> Int -> [Int]

azpiluz_lag [] luz = []

azpiluz_lag (x:s) luz
        | hutsa_da s            = (1 + luz):[]
        | x == (leh s)          = azpiluz_lag s (1 + luz)
        | x /= (leh s)          = (1 + luz): (azpiluz_lag s 0)
```

Zenbatzaileak 0tik hastea da ohikoena, baina kasu honetan luz izeneko zenbatzailea parametro orokor bat izango denez, funtzioari deitzean zenbatzaile hori geuk nahi dugun balioarekin hasiera dezakegu.

azpiluz_lag funtzioa definitu ondoren, azpiluz funtzioaren definizioa eman dezakegu; izan ere, azpiluz funtzioa azpiluz_lag funtzioaren kasu partikular bat da: luz parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrenden luzerez eratutako zerrenda itzuliko du.
-- Sarrerako datua hutsa baldin bada, zerrenda hutsa itzuliko da.
-- Definizio hau murgilketaren teknikan oinarrituta dago.

azpiluz:: [Int] -> [Int]
azpiluz r = azpiluz_lag r 0
```

Definizioan ikus daiteke azpiluz funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina azpiluz_lag funtzioa errekurtsiboa da.

azpiluz_lag funtzioak ebazten duen problema azpiluz funtzioak ebazten duena baino orokorragoa da. Izan ere, azpiluz_lag funtzioaren bidez lehenengo azpizerrendari dagokion zenbatzailea geuk nahi dugun balioarekin hasieratu dezakegu:

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 2 = [5, 2, 1, 1]

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 9 = [12, 2, 1, 1]

azpiluz funtzioa erabiliz aldiz, azpizerrenda denen luzera zehatzak kalkulatzen dira, baita lehenengo azpizerrendaren kasuan ere:

azpiluz [7, 7, 7, 20, 20, 9, 8] = [3, 2, 1, 1]

Zuzenean azpiluz_lag funtzioa erabiliz azpizerrenda denen luzera zehatzak kalkulatzea nahi badugu (lehenengo azpizerrendarena ere barne), zenbatzailea 0 balioarekin hasieratu beharko dugu:

azpiluz_lag [7, 7, 7, 20, 20, 9, 8] 0

Kasu horretan emaitza [3, 2, 1, 1] izango da.

b) Azpizerrenden zerrenda: azpizer

Zenbaki osozko zerrenda bat emanda, azpizer izeneko funtzioak jarraian dauden balio berdinez osatutako azpizerrendez eratutako zerrenda kalkulatu behar du:

azpizer [5, 5, 9, 8, 8, 8, 8, 7, 7, 8, 0, 0, 7] =

= [[5, 5], [9], [8, 8, 8, 8], [7, 7], [8], [0, 0], [7]]

azpizer [3, 3, 3, 3, 3, 3, 3, 3, 3] = [[3, 3, 3, 3, 3, 3, 3, 3, 3]]

Sarrera bezala emandako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpizer izeneko funtzioaren definizio errekurtsiboak honako egitura hau izan beharko luke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.

azpizer:: [Int] -> [[Int]]
azpizer [] = []
azpizer (x:s)
        | hutsa_da s            = [[x]]
        | x == (leh s)    = ???
        | x /= (leh s)    = ???
```

Baina kontua da nola gorde azpizerrenda bakoitza. Azpizerrendako osagaiak banan-banan eskuratu beharko dira eta horrela eratuz joan gaitezkeen azpizerrenda ez da behin betikoa izango azpizerrenda osoa bukatu arte. Gainera azpizerrenda bakoitza osorik gorde nahi da.

Azpizerrenda bakoitzaren kasuan, azpizerrendaren osagaiak gordez joan beharko dugu azpizerrenda osoa lortu arte. Azpizerrenda osoa eskuratutakoan, eraikitzen ari garen azpizerrenden zerrendan gorde dezakegu azpizerrenda hori.

Murgilketarik gabe ere egin daiteke kalkulu hori, baina ez zuzenean, beste funtzio batzuk laguntzaile moduan definitu beharko dira. Hori da orain egingo duguna. Horretarako, berdinak_hartu eta berdinak_kendu funtzio laguntzaileak definituko dira eta, gero, bi funtzio laguntzaile horiek erabiliz baina murgilketa erabili gabe, elementu berdinez osatutako azpizerrendez eratutako zerrenda kalkulatuko duen azpizer_gb funtzioaren definizioa emango da:

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_hartu:: [Int] -> [Int]

berdinak_hartu [] = error "Zerrenda hutsa."

berdinak_hartu (x:s)
        | hutsa_da s            = [x]
        | x == (leh s)          = x:(berdinak_hartu s)
        | x /= (leh s)          = [x]
```

```
-- Funtzio honek, zenbaki osozko zerrenda bat
-- emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako lehenengo azpizerrenda kenduz geratzen den zerrenda itzuliko du.
-- Sarrerako zerrenda hutsa baldin bada, errore-mezua aurkeztuko da.

berdinak_kendu:: [Int] -> [Int]

berdinak_kendu [] = error "Zerrenda hutsa."

berdinak_kendu (x:s)
        | hutsa_da s            = []
        | x == (leh s)          = berdinak_kendu s
        | x /= (leh s)          = s
```

```
-- Funtzio honek, zenbaki osozko zerrenda
-- bat emanda, zerrendan elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Kalkulua murgilketarik gabe egiten du.
-- Sarrerako zerrenda hutsa baldin bada, zerrenda hutsa itzuliko da.

azpizer_gb:: [Int] -> [[Int]]

azpizer_gb [] = []

azpizer_gb (x:s) = (berdinak_hartu (x:s)):(azpizer_gb (berdinak_kendu (x:s)))
```

azpizer_gb funtzioa erabiliz, azpizerrenda bakoitza bi aldiz zeharkatu beharko da: lehenengo aldian elementuak hartu egingo dira eta bigarren aldian kendu egingo dira. Zerrenda luzeen kasuan azpizerrenda bakoitza bi aldiz zeharkatu beharrak eraginkortasun-maila txarragoa izatea ekarriko du berarekin. Jarraian murgilketa erabiliz, azpizerrenda bakoitza behin zeharkatuko duen soluzio bat emango da.

Murgilketan oinarritutako soluzioan, azpizerrenda bakoitzaren osagaiak gordez joateko azpizerrenda bat gordetzeko balioko duen parametro berri bat behar da.

Beraz, kasu honetan murgilketa burutzeko, parametro moduan zenbaki osozko zerrenda bat edukitzeaz gain zenbaki osozko beste zerrenda bat ere izango duen funtzio laguntzailea kontsideratuko da. Bigarren parametro hori azpizerrenda

bakoitza gordez joateko erabiliko da. Azpizerrenda bakoitzarekin hasterakoan parametro hori [] balioarekin hasieratu beharko da eta azpizerrenda bakoitza bukatzean, bigarren parametro horrek une horretan duen balioa eraikitzen ari garen azpizerrenden zerrendan gorde beharko da.

Funtzio laguntzailea definitzean, azpizerrenda bat gordetzeko erabiliko den parametroak ezin du [] konstantea izan, funtzioak definitzean argumentuek parametro orokorrak izan behar baitute. Hori dela-eta, funtzio laguntzailearen esanahia honako hau izango da:

> Zenbaki osozko bi zerrenda emanda, jarraian dauden balio berdinez osatutako azpizerrendez osatutako zerrenda kalkulatuko da. Baina lehenengo azpizerrendaren kasuan azpizerrenda horri bigarren parametroaren balioa elkartuko dio eskuinetik.

Funtzio berriari azpizer_lag deituko diogu, azpizer funtzioa definitzeko laguntzaile bezala erabiliko baitugu.

```
-- Funtzio honek, zenbaki osozko bi zerrenda emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Lehenengo azpizerrendaren kasuan, azpizerrenda hori bigarren
-- parametroarekin elkartuta itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.

azpizer_lag:: [Int] -> [Int] -> [[Int]]

azpizer_lag [] azpi = []

azpizer_lag (x:s) azpi
        | hutsa_da s            = (x:azpi):[]
        | x == (leh s)          = azpizer_lag s (x:azpi)
        | x /= (leh s)          = (x:azpi):(azpizer_lag s [])
```

azpizer_lag funtzioa definitu ondoren, azpizer funtzioaren definizioa eman dezakegu; izan ere, azpizer funtzioa azpizer_lag funtzioaren kasu partikular bat da: azpi parametroaren balioa [] denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda, zerrendan
-- elkarren jarraian dauden elementu berdinez
-- osatutako azpizerrendez eratutako zerrenda itzuliko du.
-- Sarrerako lehenengo zerrenda hutsa baldin bada, zerrenda hutsa
-- itzuliko da.
-- Definizio hau murgilketaren teknikan oinarrituta dago.

azpizer:: [Int] -> [[Int]]
azpizer r = azpizer_lag r []
```

Definizioan ikus daiteke azpizer funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina azpizer_lag funtzioa errekurtsiboa da.

azpizer_lag funtzioak ebazten duen problema azpizer funtzioak ebazten duena baino orokorragoa da; izan ere, azpizer_lag funtzioaren bidez lehenengo azpizerrendari geuk nahi dugun zerrenda erantsi diezaiokegu:

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [5, 5] = [[7, 7, 7, 5, 5], [20, 20], [9], [8]]

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [3, 7, 7, 3] =

= [[7, 7, 7, 3, 7, 7, 3], [20, 20], [9], [8]]

azpizer funtzioa erabiliz aldiz, datu gisa emandako lehenengo zerrendaren azpizerrendez osatutako zerrenda itzuliko da, eta lehenengo azpizerrendaren kasuan ere ez zaio ezer erantsiko:

azpizer [7, 7, 7, 20, 20, 9, 8] = [[7, 7, 7], [20, 20], [9], [8]]

Zuzenean azpizer_lag funtzioa erabiliz datu gisa emandako lehenengo zerrendaren azpizerrendez osatutako zerrenda kalkulatzea nahi badugu (eta lehenengo azpizerrendari ezer erantsi gabe) bigarren parametroa [] balioarekin hasieratu beharko dugu:

azpizer_lag [7, 7, 7, 20, 20, 9, 8] [] = [[7, 7, 7], [20, 20], [9], [8]]

7.7.4. Tarteak zeharkatzeko indize moduan erabiliko diren parametroak eta behin-behineko emaitzak gordetzeko erabiliko diren parametroak gehitzean oinarritzen den murgilketa

Atal honetan aurkeztuko den murgilketa-adibidean bi parametro berri ipiniko dira: bata tarte bat zeharkatzeko indize gisa erabiliko da eta bestea bukaerako emaitza kalkulatzeko beharrezkoa den behin-behineko emaitza edo emaitza partziala gordetzeko erabiliko da.

a) Zenbaki beteak:

1en berdina edo handiagoa den x zenbaki bat betea dela esaten da bere zatitzaileen batura (x bera kontuan hartu gabe) x baldin bada.

Adibidez 6 betea da 1 + 2 + 3 = 6 delako eta 28 ere betea da 1 + 2 + 4 + 7 + 14 = 28 delako.

Aukera bat, dagoeneko ezagutzen ditugun *zatizer* eta *batu* izeneko funtzioak erabiltzea izango litzateke. Kasu honetan soluzioa murgilketa erabili gabe lortuko genuke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du zerrenda bateko elementuak batzen dituen
-- "batu" eta zenbaki oso baten zatitzaileen zerrenda kalkulatzen duen
-- "zatizer" funtzioak erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- Funtzio hau murgilketa erabili gabe definitu da.

betea_gb:: Int -> Bool
betea_gb x
        | x <= 0                          error "Zenbakia ez da positiboa."
        | ((batu (zatizer x)) - x) == x   = True
        | otherwise                       = False
```

Baina betea_gb funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen.

Zenbaki bat betea al den erabakitzen duen funtzio errekurtsibo bat definitzen saia gaitezke. Funtzio horrek honako egitura hau izango luke:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du errekurtsibitatea erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

betea:: Int -> Bool
betea x
        | x <= 0          = error " Zenbakia ez da positiboa."
        | otherwise       = ???
```

Baina x parametroan oinarritutako soluzio errekurtsiborik ezin da eman kasu honetan ere. Alde batetik, x zenbakiaren zatitzaileak kalkulatu behar dira. Horretarako, zatizer eta lehena izeneko funtzioetan egin den bezala, [1..x – 1] tartea zeharkatzeko indize gisa erabiliko den parametro berri bat ipini beharko da. Kontuan hartu x zenbakia betea al den erabakitzeko [1..x – 1] tarteko zatitzaileak aurkitu behar direla. Baina, gainera, x zenbakiaren tarte horretako zatitzaileen batura kalkulatu behar da, nahiz eta bukaeran itzuli beharreko emaitza batura hori ez izan. Bukaeran itzuli beharreko emaitza batura hori erabiliz lortuko da. Batura hori x zenbakiaren berdina baldin bada, orduan True itzuliko da eta bestela False itzuliko da. Beraz emaitza partzial bat kalkulatu behar da (batura) gero emaitza partzial hori erabiliz bukaerako emaitza lortu ahal izateko (azpiluz eta azpizer funtzioetan egin denaren antzera).

Guztira bi parametro berri behar dira, bata [1..x – 1] tartea zeharkatuz joateko eta bestea zatitzaileen batura gordez joateko.

Beraz, murgilketa erabiliz, [1..x – 1] tarteko zatitzaileak kalkulatu beharrean, beheko mugatzat bm edozein zenbaki izan dezakeen [bm..x – 1] tarteko zatitzaileak kalkulatuko dira. Gainera, une bakoitzean ordura arte aurkitu diren zatitzaileen batura eramango duen zatibatu izeneko parametro bat ere edukiko da. Guztira, definituko dugun funtzio berriak hiru parametro izango ditu: x, bm eta zbat. Eraginkortasuna dela-eta, x div 2 zenbakitik aurrera x zenbakiaren zatitzaile bakarra x bera izango denez, bilaketa [2..x div 2] tartera muga daiteke.

Funtzio berriari betea_lag deituko diogu, betea izeneko funtzioa definitzeko laguntzaile gisa erabiliko baitugu.

```
-- Funtzio honek, x, bm eta zatibatu hiru zenbaki oso emanda, True itzuliko du
-- x zenbakiak [bm..x - 1] tartean dituen zatitzaileen batura gehi zatibatu
-- balioa x balioaren berdina baldin bada eta bestela
-- False itzuliko du.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 0 edo negatiboa baldin bada, errorea.

betea_lag:: Int -> Int -> Int -> Bool
betea_lag x bm zatibatu
        | x <= 0                                = error "Zenbakia ez da positiboa."
        | bm <= 0                               = error "Beheko muga ez da
positiboa."
        | (bm > (x `div` 2)) && (zatibatu == x) = True
        | (bm > (x `div` 2)) && (zatibatu /= x) = False
        | (x `mod` bm) == 0                     = betea_lag x (bm + 1) (bm +
zatibatu)
        | (x `mod` bm) /= 0                     = betea_lag x (bm + 1) zatibatu
```

betea_lag funtzioa definitu ondoren, betea izeneko funtzioaren definizioa eman dezakegu; izan ere, betea funtzioa betea_lag funtzioaren kasu partikular bat da: bm parametroaren balioa 1 eta zatibatu parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- betea al den erabakiko du murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

betea:: Int -> Bool
betea x = betea_lag x 1 0
```

Definizioan ikus daiteke betea izeneko funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina betea_lag funtzioa errekurtsiboa da.

betea_lag funtzioak ebazten duen problema betea funtzioak ebazten duena baino orokorragoa da. Izan ere, betea_lag funtzioaren bidez 25 zenbakiaren [3..24] tarteko zatitzaileen batura gehi 20 balioa 25 al den jakin dezakegu:

```
betea_lag 25 3 20
```

Erantzuna True izango litzateke; izan ere, [3..24] tartean 25 zenbakiaren zatitzaile bakarra daukagu (5) eta zatitzaile horri 20 batuz 25 balioa lortzen da.

Esate baterako, betea_lag 25 3 15 deia eginez gero, False erantzuna jasoko genuke.

Era berean, betea_lag 77 5 30 deia eginez ere erantzuna False izango litzateke, [5..76] tartekoak diren 77ren zatitzaileen batura (7 + 11) gehi 30 ez baita 77. Baina betea_lag 77 5 59 deiak True balioa itzuliko luke 7 + 11 + 59 baturaren balioa 77 baita.

Bestalde, betea_lag 28 4 3 deiak ere True itzuliko luke 4 + 7 + 14 + 3 = 28 baita, baina betea_lag 28 4 0 deiaren erantzuna False izango litzateke 4 + 7 + 14 + 0 baturaren emaitza ez baita 28. Beste kasu bat aztertuz, lehena_lag 28 6 3 deiaren erantzuna ere False izango litzateke 7 + 14 + 3 baturaren balioa ez baita 28.

Gogoratu betea izeneko funtzioarekin zatitzaileak beti 1etik hasita bilatuko direla eta baturaren hasierako balioa 0 izango dela:

betea 77 = False
betea 28 = True
betea 25 = False

Zuzenean betea_lag erabiliz 28 betea al den erabakitzea nahi badugu, honako hau idatzi beharko dugu:

betea_lag 28 1 0

7.7.5. Murgilketaren beste aplikazio batzuk: bukaerako errekurtsibitatea eta eraginkortasuna

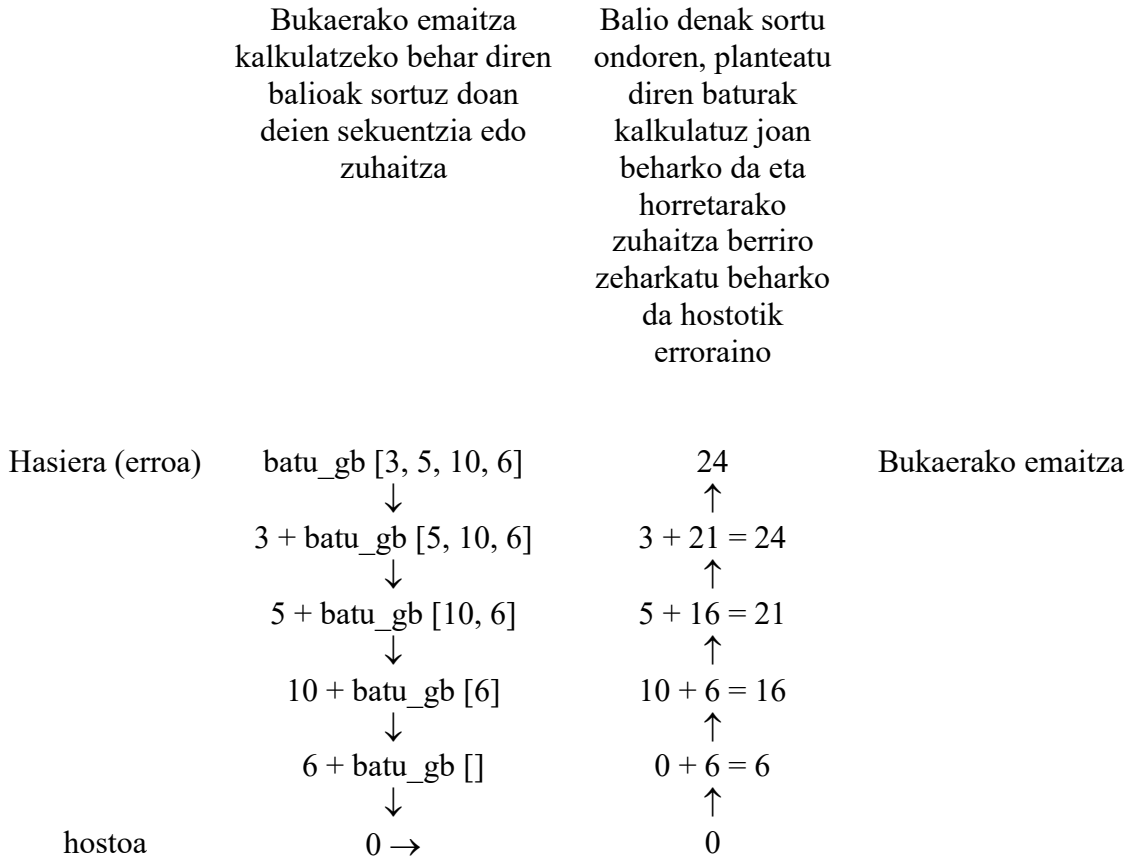
Aurreko ataleko adibideetan ikusi da batzuetan ez dela beharrezkoa murgilketa erabiltzea baina hala ere erabil daitekela, murgilketarik gabe definitutako funtzioak baino eraginkorragoak diren funtzioak lortuz. Atal honetan oraindik eraginkortasun handiagoa lortzeko beste urrats bat emango dugu eta *bukaerako errekurtsibitatea* duten soluzioak murgilketa erabiliz nola lortu ditzakegun azalduko da. Bukaerako errekurtsibitatea edukitzeak eraginkortasun-maila hobetzen du oro har.

a) Zerrenda bateko elementuen batura:

Zenbaki osozko zerrenda bateko elementuen batura era errekurtsiboan kalkulatzen duen funtzioa honela defini daiteke:

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- zerrendako elementuen batura kalkulatuko du
-- murgilketaren teknika erabili gabe.

batu_gb :: [Int] -> Int

batu_gb [] = 0

batu_gb (x:s) = x +  (batu_gb s)
```

Definizio horretan ez da murgilketa erabili. Zerrenda bateko elementuen batura kalkulatzen duen funtzio horrek ez du *bukaerako errekurtsibitaterik*, dei errekurtsiboei dagokien sekuentzia edo zuhaitza garatu ondoren, garapen horretan sortuz joan diren elementuak batzeko zuhaitza berriro zeharkatu behar baita kontrako eran:



batu_gb [3, 5, 10, 6] deiak adar bakarreko zuhaitz bat sortuko du eta adabegi bakoitzean emaitza partzial bat lagako da (zenbaki bat). Azkeneko adabegira, hau da, 0 balioa duen adabegira iritsitakoan (adabegi hori hostoa da), zuhaitza berriro hostotik erroraino zeharkatu beharko da, adabegi bakoitzean lagatako zenbakia jasoz eta batura kalkulatuz. Errora iritsitakoan behin betiko emaitza edukiko da.

Beraz, zuhaitza bi aldiz zeharkatu behar da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun galera sortuko du.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joatean adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzean, hau da, errotik hostora joatean bukaerako emaitza eraikiz eta behera eramanez joan gaitezke. Eraikitzen ari garen emaitza hori beheruntz pasatzeko batura gordez doan parametro berri bat erabiliko genuke.

Parametro gisa batu beharreko elementuez osatutako zerrenda edukitzeaz gain, batura kalkulatuz joateko erabiliko den beste parametro bat ere baduen batu_lag izeneko funtzioa definitiko dugu orain:

```
-- Funtzio honek, zenbaki osozko zerrenda bat eta b zenbaki oso bat emanda,
-- zerrendako elementuen batura gehi b kalkulatuko du.

batu_lag :: [Int] -> Int -> Int

batu_lag [] b = b

batu_lag (x:s) b = batu_lag s (x + b)
```

batu_lag funtzioa definitu ondoren, batu_mr izeneko funtzioaren definizioa eman dezakegu; izan ere, batu_mr funtzioa batu_lag funtzioaren kasu partikular bat da: b parametroaren balioa 0 denekoa hain zuzen ere.

```
-- Funtzio honek, zenbaki osozko zerrenda bat emanda,
-- zerrendako elementuen batura kalkulatuko du
-- murgilketaren teknika erabiliz.

batu_mr :: [Int] -> Int

batu_mr r = batu_lag r 0
```

Definizioan ikus daitekeen bezala, batu_mr izeneko funtzioa berez ez da errekurtsiboa, ez baitio bere buruari deitzen, baina batu_lag funtzioa errekurtsiboa da.

batu_lag funtzioak ebazten duen problema batu_mr funtzioak ebazten duena baino orokorragoa da; izan ere, batu_lag funtzioaren bidez zerrenda bateko elementuen batura kalkulatzeaz gain batura horri beste balio bat ere batu diezaiokegu:

batu_lag [10, 4, 5] 6

deiaren kasuan erantzuna 25 izango litzateke (19 + 6).

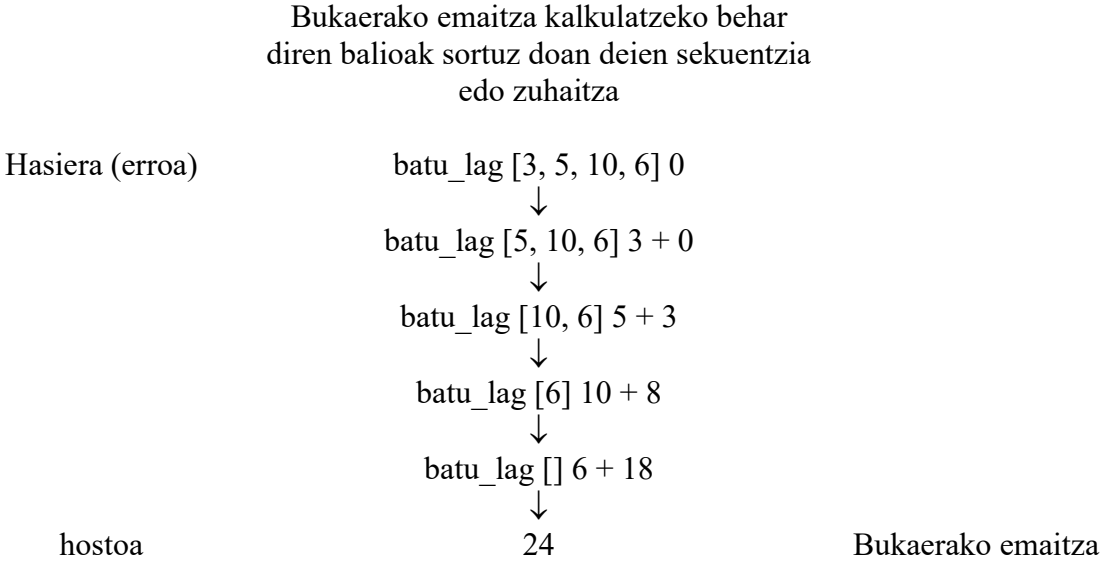
Bestalde, batu_mr funtzioak zerrendako elementuen batura kalkulatuko du, beste ezer gehitu gabe:

batu_mr [10, 4, 5] = 19

Zuzenean batu_lag funtzioa erabiliz [10, 4, 5] zerrendako elementuen batura kalkulatu nahi badugu, honako hau ipini beharko dugu:

batu_lag [10, 4, 5] 0

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratuko denez (zuhaitza berriro zeharkatu beharrik gabe), batu_lag funtzioak bukaerako errekurtsibitatea du:



batu_lag [3, 5, 10, 6] 0 deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzean ordura arte tratatu diren elementuen batura edukiko da. Azkeneko adabegira iritsitakoan (24 balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

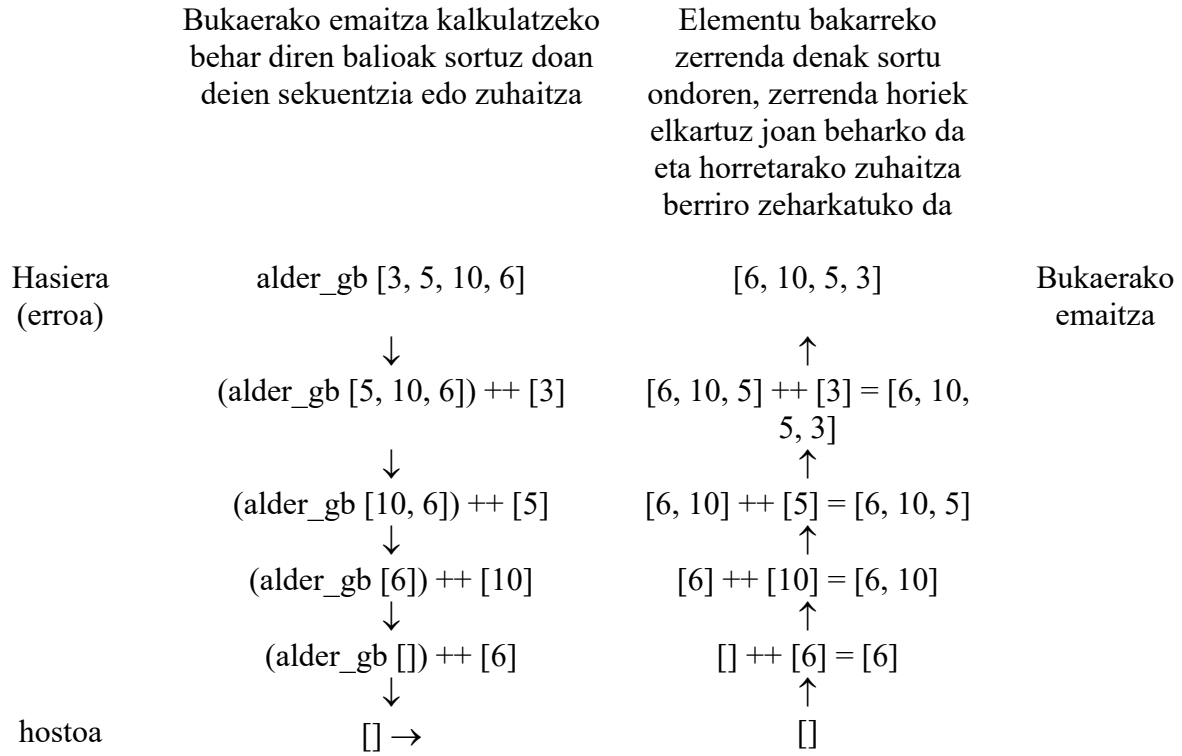
Beraz batu_lag funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den batu_gb funtzioa baino eraginkorragoa da.

b) Zerrenda baten alderantzizkoa.

Zerrenda baten alderantzizkoa era errekurtsiboan honela kalkula daiteke:

```
-- Funtzio honek, zerrenda bat emanda, bere alderantzizkoa itzuliko
-- du ++ erabiliz eta murgilketa erabili gabe.

alder_gb :: [t] -> [t]

alder_gb [] = []

alder_gb (x:s) = (alder_gb s) ++ [x]
```

Definizio horretan ez da murgilketa erabili. Zerrenda baten alderantzizkoa kalkulatuko duen funtzio horrek ez du bukaerako errekurtsibitaterik, beharrezkoak diren dei errekurtsiboz osatutako sekuentzia (edo zuhaitza) eraiki ondoren, behin betiko emaitza kalkulatzeko adabegi bakoitzean sortu diren elementu bakarreko zerrendak elkartu behar baitira eta, horretarako, zuhaitza berriro zeharkatu behar da behetik gora:



alder_gb [3, 5, 10, 6] deia adar bakarreko zuhaitz bat eraikiz joango da eta adabegi bakoitzean emaitza partzial bat lagako da. Azkeneko adabegira iritsitakoan, hau da, [] balioa duen eta hostoa den adabegira iritsitakoan, adabegi bakoitzean lagatako elementu bakarreko zerrenda jasotzeko eta behin betiko zerrenda eraikiz joateko zuhaitz dena berriro zeharkatu beharko da. Errora iritsitakoan behin betiko zerrenda edukiko da.

Beraz, zuhaitza bi aldiz zeharkatu beharko da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun-galera ekarriko du berarekin.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joatean adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzean, hau da, errotik hostora joatean bukaerako emaitza eraikiz joan gaitezke. Eraikiz ari garen emaitza hori beherantz pasatuz joango ginateke alderantzizko zerrenda gordez doan parametro berri baten bidez.

Parametro gisa alderantziz ipini beharreko zerrenda edukitzeaz gain alderantzizko zerrenda kalkulatuz joateko erabiliko den beste parametro bat ere baduen alder_lag izeneko funtzioa definituko dugu orain:

```
-- Funtzio honek, bi zerrenda emanda, lehenengo zerrendaren
-- alderantzizkoa kalkulatuko du eta
-- bigarren zerrendari elkartuta itzuliko du.

alder_lag :: Show t => [t] -> [t] -> [t]

alder_lag [] q = q

alder_lag (x:s) q = alder_lag s (x:q)
```

alder_lag funtzioa definitu ondoren, alder_mr izeneko funtzioaren definizioa eman dezakegu; izan ere, alder_mr funtzioa alder_lag funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere.

```
-- Funtzio honek, zerrenda bat emanda, bere alderantzizkoa itzuliko
-- du murgilketa erabiliz.

alder_mr :: [t] -> [t]

alder_mr r = alder_lag r []
```

Definizioan ikus daiteke alder_mr izeneko funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina alder_lag funtzioa errekurtsiboa da.

alder_lag funtzioak ebazten duen problema alder_mr funtzioak ebazten duena baino orokorragoa da; izan ere, alder_lag funtzioaren bidez lehenengo zerrendaren alderantzizkoari beste zerrenda bat erantsi diezaiokegu:

alder_lag [2, 8, 9] [4, 5]

Erantzuna [9, 8, 2, 4, 5] izango litzateke.

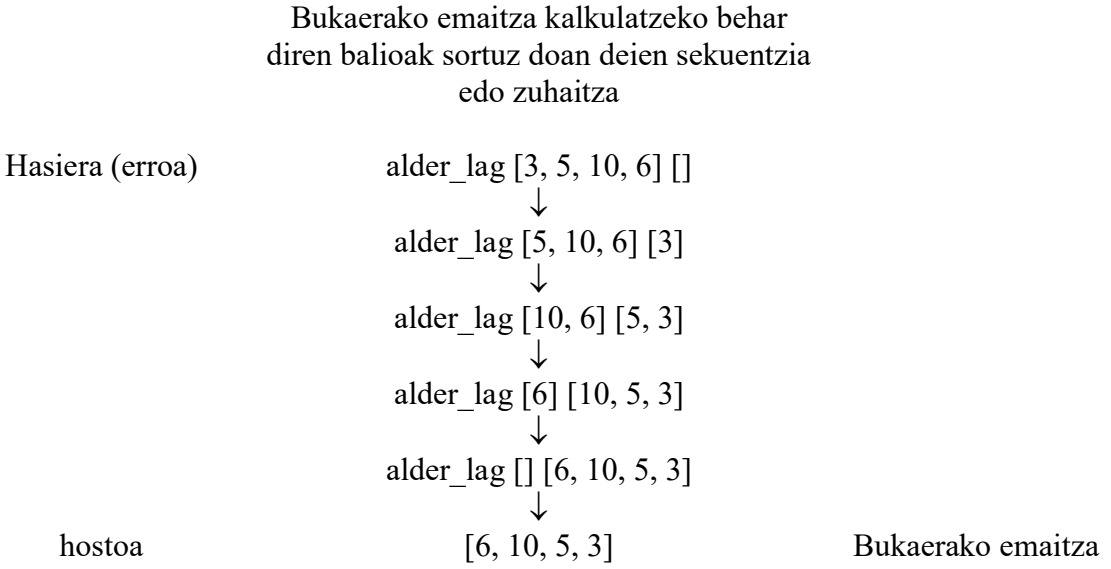
Baina alder_mr funtzioarekin zerrendaren alderantzizkoa kalkulatuko da, beste ezer erantsi gabe:

alder_mr [2, 8, 9] = [9, 8, 2]

alder_lag erabiliz [2, 8, 9] zerrendaren alderantzizkoa kalkulatzea nahi izanez gero, honako hau ipini beharko genuke:

alder_lag [2, 8, 9] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratzen denez (zuhaitza berriro zeharkatu beharrik gabe), alder_lag funtzioak bukaerako errekurtsibitatea du:



alder_lag [3, 5, 10, 6] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzean ordura arte tratatu den zerrenda zatiaren alderantzizkoa edukiko da. Azkeneko adabegira iritsitakoan ([6, 10, 5, 3] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro gorantz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

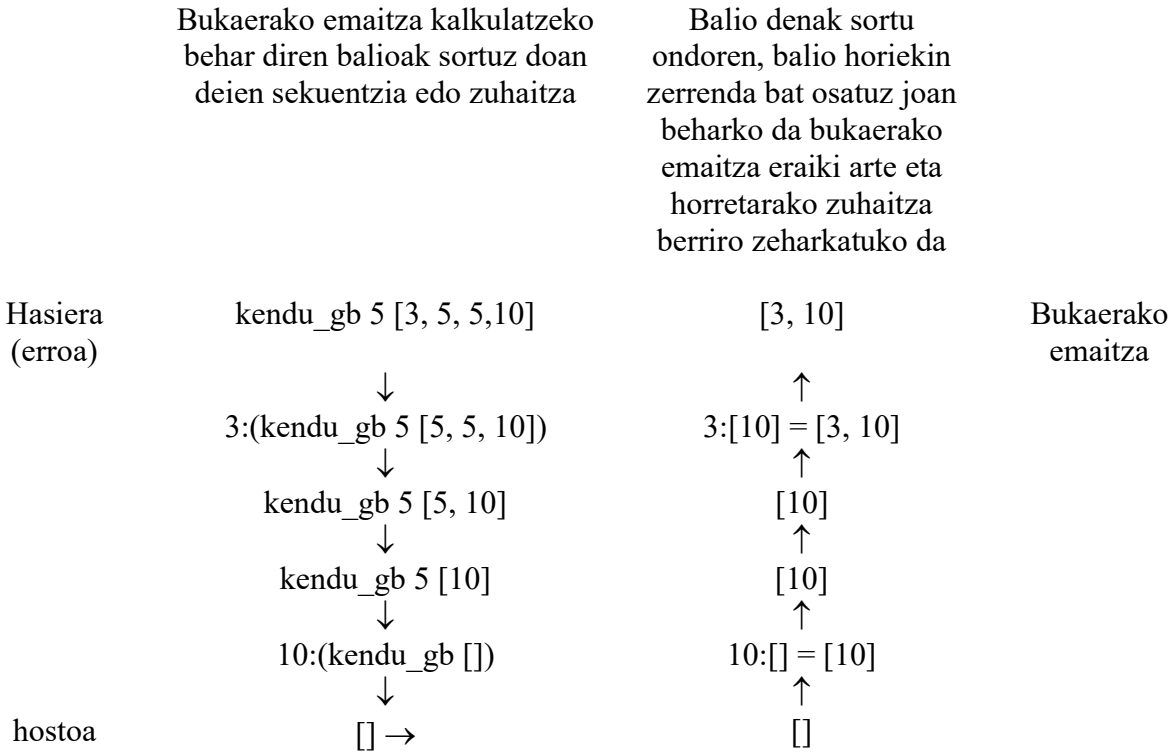
Beraz, alder_lag funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den alder_gb funtzioa baino eraginkorragoa da.

c) Elementu baten agerpen denak zerrendatik kentzen dituen funtzioa.

Elementu bat eta zerrenda bat emanda, zerrendatik elementu horren agerpen denak ezabatuko dituen funtzioa honela defini daiteke errekurtsiboki:

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz gelditzen den zerrenda itzuliko du
-- murgilketa erabili gabe.

kendu_gb :: Eq t => t -> [t] -> [t]

kendu_gb x [] = []

kendu_gb x (y:s)
        | x == y        = kendu_gb x s
        | x /= y        = y:(kendu_gb x s)
```

Definizio horretan ez da murgilketa erabili. Zerrendatik elementu baten agerpen denak ezabatzen dituen funtzio horrek ez du *bukaerako errekurtsibitaterik*, dei errekurtsiboei dagokien sekuentzia edo zuhaitza garatu ondoren, garapen horretan geratuz joan diren elementuekin emaitza osatzeko zuhaitza berriro zeharkatu behar baita kontrako eran:



kendu_gb 5 [3, 5, 5, 10] deia adar bakarreko zuhaitz bat eraikiz joango da eta adabegi bakoitzean emaitza partzial bat lagako da. Azkeneko adabegira iritsitakoan, hau da, [] balioa duen eta hostoa den adabegira iritsitakoan, adabegi bakoitzean lagatako elementua jasotzeko eta behin betiko zerrenda eraikiz joateko zuhaitz dena berriro zeharkatu beharko da. Errora iritsitakoan behin betiko zerrenda edukiko da.

Beraz, zuhaitza bi aldiz zeharkatu beharko da. Zuhaitza oso sakona baldin bada, bi aldiz zeharkatu beharrak eraginkortasun-galera ekarriko du berarekin.

Hostora iritsitakoan bukaerako emaitza edukitzea izango litzateke onena. Horrela zuhaitza bigarren aldiz (oraingoan hostotik errora) zeharkatu beharrik ez genuke izango eta funtzioa eraginkorragoa izango litzateke.

Hori murgilketa erabiliz lor daiteke. Errotik hostora joatean adabegietan balioak lagaz joan eta gero bukaerako emaitza eraikitzeko zuhaitza berriro (oraingoan hostotik errora) zeharkatu beharrean, zuhaitza eraikitzean, hau da, errotik hostora joatean, bukaerako emaitza eraikiz joan gaitezke. Eraikiz ari garen emaitza hori beherantz pasatuz joango ginateke zerrenda gordez doan parametro berri baten bidez.

Parametro gisa, kendu beharreko elementua eta zerrenda bat edukitzeaz gain, elementu horren agerpenak zerrendatik kenduz lortuko den zerrenda kalkulatuz joateko erabiliko den beste parametro bat ere baduen kendu_lag_ald izeneko funtzioa definituko dugu orain:

```
-- Funtzio honek, elementu bat eta bi zerrenda emanda, elementu horren
-- agerpen denak lehenengo zerrendatik kenduz gelditzen den zerrenda
-- eta aurretik (ezkerretik) bigarren zerrendaren alderantzizkoa
-- elkartuz lortzen den zerrenda itzuliko du.
-- Aurretik definitutako alder_mr funtzioa erabiliko da.

kendu_lag_ald :: Eq t => t -> [t] -> [t] -> [t]

kendu_lag_ald x [] q = alder_mr q

kendu_lag_ald x (y:s) q
        | x == y        = kendu_lag_ald x s q
        | x /= y        = kendu_lag_ald x s (y:q)
```

Funtzioaren definizioa edozein t motarentzat planteatu da eta funtzioa definitzen duten ekuazioetan berdintasuna (==) eta desberdintasuna (/=) erabili dira. Funtzioak t mota bateko elementuentzat zentzua izan dezan, t mota horretan berdintasunak eta desberdintasunak definituta egon behar dutela zehazten da Eq t => ipiniz.

kendu_lag_ald funtzioa definitu ondoren, kendu_mr_ald izeneko funtzioaren definizioa eman dezakegu. Izan ere, kendu_mr_ald funtzioa kendu_lag_ald funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere.

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz geldituko den zerrenda itzuliko du
-- zerrenda baten alderantzizko kalkulatzen duen funtzioan oinarritutako
-- murgilketa erabiliz.

kendu_mr_ald :: Eq t => t -> [t] -> [t]

kendu_mr_ald x r = kendu_lag_ald x r []
```

Definizioan ikus daiteke kendu_mr_ald izeneko funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina kendu_lag_ald funtzioa errekurtsiboa da.

kendu_lag_ald funtzioak ebazten duen problema kendu_mr_ald funtzioak ebazten duena baino orokorragoa da. Izan ere, kendu_lag_ald funtzioaren bidez lehenengo zerrendatik elementuaren agerpen denak kentzeaz gain, beste zerrenda baten alderantzizkoa erantsi diezaiokegu ezkerretik:

```
kendu_lag_ald 5 [3, 5, 5, 10] [0, 5, 2]
```

erantzuna [2, 5, 0, 3, 10] izango litzateke.

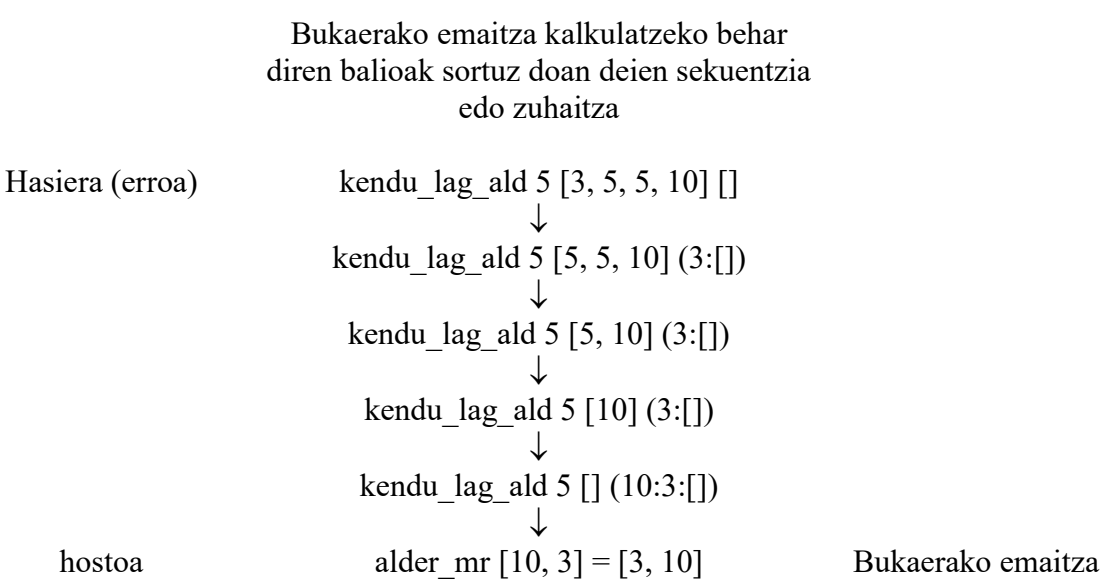
Baina kendu_mr_ald funtzioarekin, emandako elementuaren agerpen denak zerrendatik kenduz geratzen den zerrenda kalkulatuko da, beste ezer erantsi gabe:

kendu_mr_ald 5 [3, 5, 5, 10] = [3, 10]

Zuzenean kendu_lag_ald erabiliz [3, 5, 5, 10] zerrendatik 5 zenbakiaren agerpen denak kenduz geratzen den zerrenda lortu nahi bada, honako hau ipini beharko da:

kendu_lag_ald 5 [3, 5, 5, 10] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratuko denez (zuhaitza berriro zeharkatu beharrik gabe), kendu_lag_ald funtzioak bukaerako errekurtsibitatea du:



kendu_lag_ald 5 [3, 5, 5, 10] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzean ordura arte tratatu den zerrenda zatitik hartu beharreko elementuz osatutako zerrendaren alderantzizkoa edukiko da. Azkeneko adabegira iritsitakoan ([3, 10] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu. Adibidean ikusten da prozesuan eraikiz joan garen zerrendaren alderantzizkoa itzuli behar dela bukaerako emaitza gisa.

Beraz kendu_lag_ald funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den kendu_gb funtzioa baino eraginkorragoa da.

BESTE AUKERA BAT:

Murgilketa aplikatuz baina alderantzizkoa kalkulatzen duen funtzioa erabili gabe ere defini daiteke elementu baten agerpen denak zerrendatik kenduko dituen funtzio bat. Kasu honetan bi zerrenda elkartzeko balio duen (++) funtzioa erabiliko da.

Parametro gisa, kendu beharreko elementua eta zerrenda bat edukitzeaz gain elementu horren agerpenak zerrendatik kenduz lortuko den zerrenda kalkulatuz

joateko erabiliko den beste parametro bat ere baduen kendu_lag_elk izeneko funtzioa definituko dugu orain:

```
-- Funtzio honek, elementu bat eta bi zerrenda emanda, elementu horren
-- agerpen denak lehenengo zerrendatik kenduz gelditzen den zerrenda
-- eta aurretik (ezkerretik) bigarren zerrenda
-- elkartuz lortzen den zerrenda itzuliko du.
-- ++ funtzioa erabiliko da.

kendu_lag_elk :: Eq t => t -> [t] -> [t] -> [t]

kendu_lag_elk x [] q = q

kendu_lag_elk x (y:s) q
        | x == y        = kendu_lag_elk x s q
        | x /= y        = kendu_lag_elk x s (q ++ [y])
```

Funtzioaren definizioa edozein t motarentzat planteatu da eta funtzioa definitzen duten ekuazioetan berdintasuna (==) eta desberdintasuna (/=) erabili dira. Funtzioak t mota bateko elementuentzat zentzua izan dezan, t mota horretan berdintasunak eta desberdintasunak definituta egon behar dutela zehazten da Eq t => ipiniz.

kendu_lag_elk funtzioa definitu ondoren, kendu_mr_elk izeneko funtzioaren definizioa eman dezakegu; izan ere, kendu_mr_elk funtzioa kendu_lag_elk funtzioaren kasu partikular bat da: q parametroaren balioa [] denekoa hain zuzen ere

```
-- Funtzio honek, elementu bat eta zerrenda bat emanda, elementu horren
-- agerpen denak kenduz gelditzen den zerrenda itzuliko du
-- bi zerrenda elkartzen dituen ++ eragiketan oinarritutako murgilketa
-- erabiliz.

kendu_mr_elk :: Eq t => t -> [t] -> [t]

kendu_mr_elk x r = kendu_lag_elk x r []
```

Definizioan ikus daiteke kendu_mr_elk izeneko funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina kendu_lag_elk funtzioa errekurtsiboa da.

kendu_lag_elk funtzioak ebazten duen problema kendu_mr_elk funtzioak ebazten duena baino orokorragoa da; izan ere, kendu_lag_elk funtzioaren bidez lehenengo zerrendatik elementuaren agerpen denak kentzeaz gain, beste zerrenda bat erantsi diezaiokegu ezkerretik:

```
kendu_lag_elk 5 [3, 5, 5, 10] [0, 5, 2]
```

Erantzuna [0, 5, 2, 3, 10] izango litzateke.

Baina kendu_mr_elk funtzioarekin, emandako elementuaren agerpen denak zerrendatik kenduz geratzen den zerrenda kalkulatuko da, beste ezer erantsi gabe:

kendu_mr_elk 5 [3, 5, 5, 10] = [3, 10]

Zuzenean kendu_lag_elk erabiliz [3, 5, 5, 10] zerrendatik 5 zenbakiaren agerpen denak kenduz geratzen den zerrenda lortu nahi bada, honako hau ipini beharko da:

kendu_lag_elk 5 [3, 5, 5, 10] []

Dei errekurtsiboez osatutako sekuentzia edo zuhaitza eraikitakoan emaitza kalkulatuta geratuko denez (zuhaitza berriro zeharkatu beharrik gabe), kendu_lag_elk funtzioak bukaerako errekurtsibitatea du:

Bukaerako emaitza kalkulatzeko behar
diren balioak sortuz doan deien sekuentzia
edo zuhaitza

Hasiera (erroa) kendu_lag_elk 5 [3, 5, 5, 10] []
↓
kendu_lag_elk 5 [5, 5, 10] ([] ++ [3])
↓
kendu_lag_elk 5 [5, 10] [3]
↓
kendu_lag_elk 5 [10] [3]
↓
kendu_lag_elk 5 [] ([3] ++ [10])
↓
hostoa [3, 10] Bukaerako emaitza

kendu_lag_elk 5 [3, 5, 5, 10] [] deiak adar bakarra duen zuhaitz baten eraketari hasiera emango dio. Zuhaitz horretako adabegi berri bakoitza sortzean ordura arte tratatu den zerrenda zatitik hartu beharreko elementuz osatutako zerrenda edukiko da. Azkeneko adabegira iritsitakoan ([3, 10] balioa duena eta zuhaitz horrentzat hostoa dena) ez dago zuhaitza berriro goruntz zeharkatu beharrik, bukaerako emaitza hor bertan baitaukagu.

Beraz kendu_lag_elk funtzioarekin zuhaitza behin bakarrik zeharkatuko da eta ondorioz lehen definitu den kendu_gb funtzioa baino eraginkorragoa da.

d) Fibonacci-a: *fib*

0ren fibonaccia 0 da, 1en fibonaccia 1 da eta 2 edo handiagoa den x zenbaki baten fibonaccia x – 1 eta x – 2 zenbakien fibonacciak batuz lortzen da.

Adibideak:

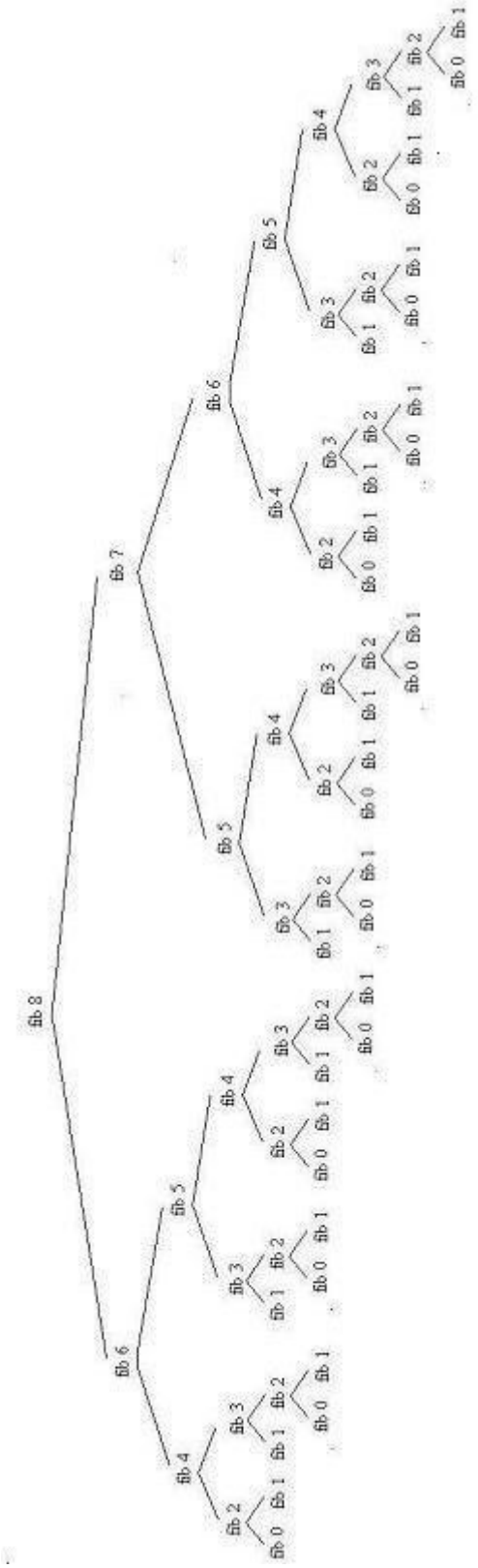
fib 2 = fib 0 + fib 1 = 0 + 1 = 1

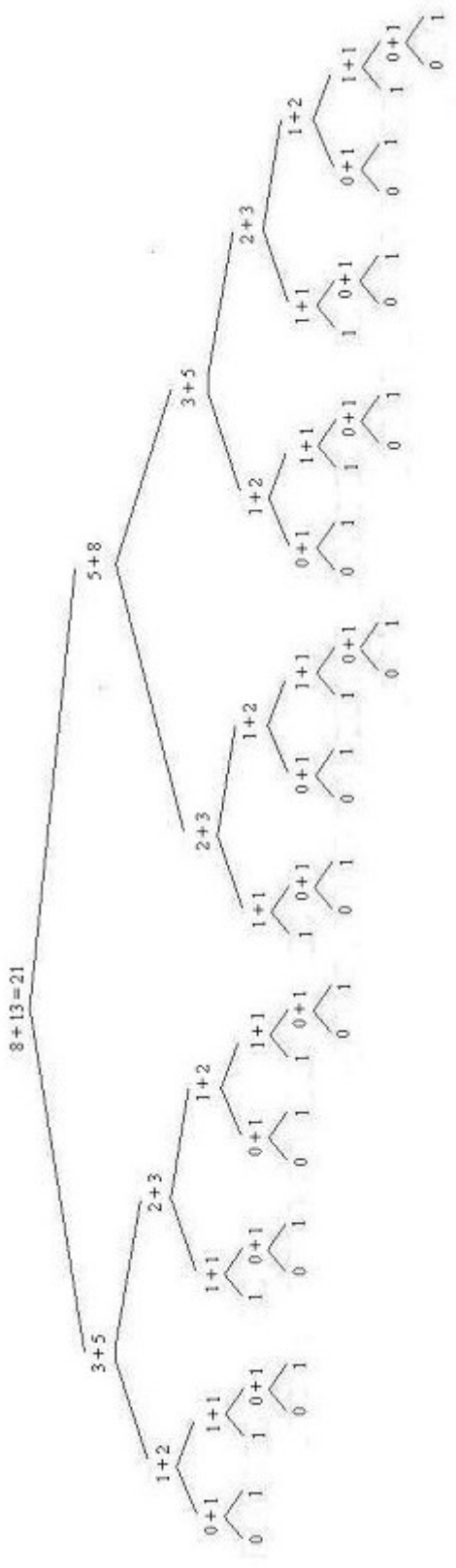
fib 3 = fib 1 + fib 2 = 1 + 1 = 2
fib 4 = fib 2 + fib 3 = 1 + 2 = 3
fib 5 = fib 3 + fib 4 = 2 + 3 = 5
fib 6 = fib 4 + fib 5 = 3 + 5 = 8
fib 7 = fib 5 + fib 6 = 5 + 8 = 13
fib 8 = fib 6 + fib 7 = 8 + 13 = 21

Honaho funtzio honek x zenbakiaren fibonaccia kalkulatzen du murgilketa erabili gabe.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabili gabe.
-- x zenbakia negatiboa baldin bada, errorea.

fib:: Int -> Int
fib x
        | x <= (-1)             = error "Zenbaki negatiboa."
        | x == 0                = 0
        | x == 1                = 1
        | otherwise             = (fib (x - 1)) + (fib (x - 2))
```

Definizio horretan ez da murgilketarik erabili. Fibonaccia kalkulatzen duen funtzio horrek ez du *bukaerako errekurtsibitaterik*, dei errekurtsiboei dagokien zuhaitza garatu ondoren, garapen horretan agertuz joan diren elementuekin emaitza osatzeko zuhaitza berriro zeharkatu behar baita kontrako eran, hostoetatik erroraino. Beraz, zuhaitza bigarren aldiz zeharkatu behar da.


fib 8
fib 6
fib 7
fib 4
fib 5
fib 5
fib 6
fib 2
fib 3
fib 3
fib 4
fib 4
fib 5
fib 0 fib 1
fib 1 fib 2
fib 2 fib 3
fib 3 fib 4
fib 3 fib 4
fib 4 fib 5


8 + 13 = 21
3 + 5
5 + 8
1 + 2
2 + 3
2 + 3
3 + 5
0 + 1
1 + 1
1 + 1
1 + 2
1 + 2
2 + 3

Aurreko orrialdeko irudian, 8ren fibonaccia kalkulatzeko behar diren deiez osatutako zuhaitza ikus daiteke. Zuhaitz hori errotik hasita eraikitzen da (fib 8 ipintzen duen lekutik) eta beherantz joan behar da hostoetara iritsi arte (fib 0 eta fib 1 duten adabegietaraino). Fibonaccia kalkulatzean oinarrizko balioak fib 0 eta fib 1 direnez (0 eta 1 hurrenez hurren), hostoetatik abiatuz batura kalkulatu beharko da. Beraz, baturen kalkulua egitean, zuhaitza hostoetatik errora zeharkatuko da. Errora iritsitakoan bukaerako batura edukiko da, hau da, 8ren fibonaccia.

Zuhaitza bi aldiz zeharkatu behar izateaz gain, kalkulu batzuk errepikatuta daudela ere ikus dezakegu:

- 6ren fibonaccia 2 aldiz kalkulatzen da.
- 5en fibonaccia 3 aldiz kalkulatzen da.
- 4ren fibonaccia 5 aldiz kalkulatzen da.
- 3ren fibonaccia 8 aldiz kalkulatzen da.
- 2ren fibonaccia 13 aldiz kalkulatzen da.
- 1en fibonaccia 21 aldiz kalkulatzen da.
- 0ren fibonaccia 13 aldiz kalkulatzen da.

Kalkuluak hainbeste aldiz errepikatu behar izateak, eraginkortasun-galera dakar. Murgilketa erabiliz kalkuluak hainbeste aldiz ez errepikatzea eta zuhaitza behin bakarrik zeharkatzea lortuko dugu.

2 edo handiagoa den x zenbakiaren fibonaccia murgilketa erabiliz honela kalkula daiteke:

- 2 zenbakitik abiatu eta [2..x] tarteko zenbakiak zeharkatuz joan.
- Tartea zeharkatu ahala, zenbaki bakoitzaren fibonaccia kalkulatuz joan (2rena, 3rena, 4rena, …)
- Une bakoitzean zein zenbakitan gauden jakiteko, hau da, beheko muga une bakoitzean zein den jakiteko, bm izeneko parametro berri bat erabiliko da.
- Zenbaki baten fibonaccia kalkulatzeko aurreko bi zenbakien fibonaccia ezagutu behar denez, bm zenbakiaren aurreko bi zenbakien (hau da, bm – 1 eta bm – 2 zenbakien) fibonaccia gordeta edukiko dugu beste bi parametro berriren bidez, aa eta aur (aurrekoaren aurrekoa eta aurrekoa)

Funtzio berriari fib_lag deituko diogu. Izan ere, fibonaccia kalkulatzen duen funtzioa murgilketaren bidez kalkulatzeko laguntzaile gisa erabiliko dugu.

```
-- Funtzio honek, x, bm, aa eta aur lau zenbaki oso emanda,
-- [bm..x] tartea zeharkatuko du.
-- x zenbakia negatiboa baldin bada, errorea.
-- bm-ren balioa 1 edo txikiagoa baldin bada, errorea.

fib_lag:: Int -> Int -> Int -> Int -> Int
fib_lag x bm aa aur
        | x <= (-1)             = error "Zenbaki negatiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 0 || x == 1      = x
        | bm > x                = error "Eremu hutsa."
        | bm == x               = aa + aur
        | otherwise             = fib_lag x (bm + 1) aur (aa + aur)
```

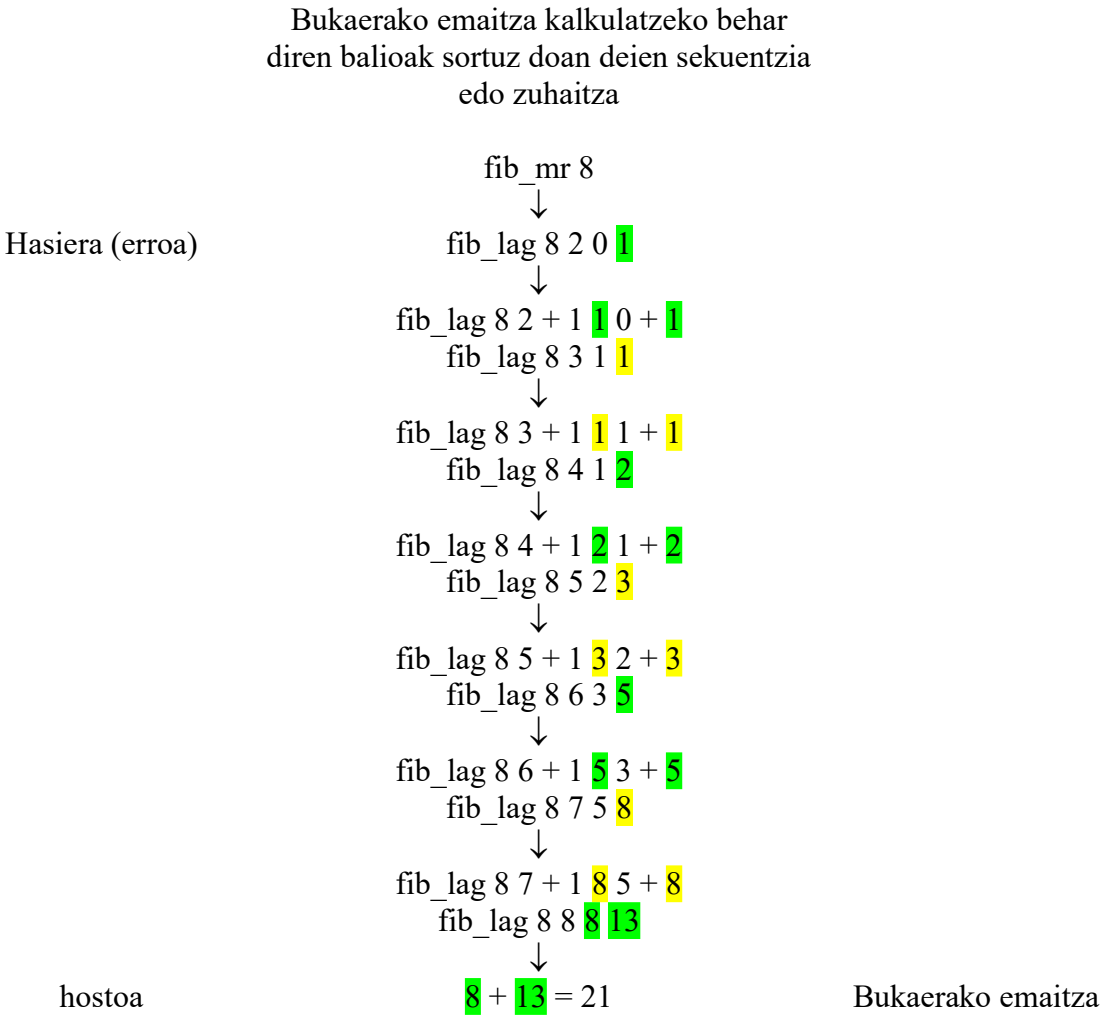
fib_lag funtzioa definitu ondoren, fib_mr izeneko funtzioaren definizioa eman dezakegu. Izan ere, fib_mr funtzioa fib_lag funtzioaren kasu partikular bat da: bm, aa eta aur parametroen balioak hurrenez hurren 2, 0 eta 1 direnekoa hain zuzen ere.

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabiliz.
-- x zenbakia negatiboa baldin bada, errorea.

fib_mr:: Int -> Int

fib_mr x = fib_lag x 2 0 1
```

Definizioan ikus daiteke fib_mr izeneko funtzioa berez ez dela errekurtsiboa, ez baitio bere buruari deitzen, baina fib_lag funtzioa errekurtsiboa da.

fib_lag funtzioak ebazten duen problema fib_mr funtzioak ebazten duena baino orokorragoa da.

fib_mr erabiliz 8ren fibonaccia nola kalkulatuko genukeen azalduko da jarraian:



Adibide honekin, murgilketaren bidez lortutako soluzioaren bidez askoz adabegi gutxiago dituen eta adar bakarra duen zuhaitza eraikitzen dela ikus dezakegu. Gainera ez du kalkulurik errepikatzen. Ondorioz, askoz eraginkorragoa da.

Int mota mugatua da. Int motaren barruan adieraz daitekeen balio txikiena zein den jakiteko honako hau idatzi beharko dugu:

```
minBound :: Int
```

Eta balio handiena zein den jakiteko honako hau idatzi beharko dugu:

```
maxBound :: Int
```

Int mota mugatua denez, fib, fib_lag eta fib_mr funtzioak, nahiko txikiak diren x parametroaren balioentzat erantzun ezinda gelditzen dira edo zentzurik ez duen erantzuna itzultzen dute (adibidez balio negatibo bat). Hori dela eta, Integer mota erabiltzen duten bertsioak defini ditzakegu. Integer motak ez du mugarik eta zenbaki oso denak adieraz daitezke mota horretan. Integer mota erabiliz idatzi diren bertsio hauei izenean 2 erantsi zaie:

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabili gabe.
-- x zenbakia negatiboa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2:: Integer -> Integer
fib2 x
        | x <= (-1)             = error "Zenbaki negatiboa."
        | x == 0                = 0
        | x == 1                = 1
        | otherwise             = (fib2 (x - 1)) + (fib2 (x - 2))
```

```
-- Funtzio honek, x, bm, aa eta aur lau zenbaki oso emanda,
-- [bm..x] tartea zeharkatuko du.
-- x zenbakia negatiboa baldin bada, errorea.
-- bm-ren balioa 1 edo txikiagoa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2_lag:: Integer -> Integer -> Integer -> Integer -> Integer
fib2_lag x bm aa aur
        | x <= (-1)             = error "Zenbaki negatiboa."
        | bm <= 1               = error "Beheko muga 2 baino txikiagoa."
        | x == 0 || x == 1      = x
        | bm > x                = error "Eremu hutsa."
        | bm == x               = aa + aur
        | otherwise             = fib2_lag x (bm + 1) aur (aa + aur)
```

```
-- Funtzio honek, x zenbaki oso bat emanda,
-- bere fibonaccia kalkulatuko du murgilketa erabiliz.
-- x zenbakia negatiboa baldin bada, errorea.
-- Int mota erabili beharrean Integer mota erabili da.

fib2_mr:: Integer -> Integer

fib2_mr x = fib2_lag x 2 0 1
```

Integer mota erabiliz idatzitako bertsio hauen bidez, fib2_mr funtzioa fib2 funtzioa baino eraginkorragoa dela ikus daiteke. Horretarako nahikoa da, esate baterako, honako kasu hauekin frogatzea:

```
fib2 40
fib2_mr 40

fib2 30
fib2_mr 30
```

eta abar.

7.8. Aurredefinitutako funtzio batzuk zerrendentzat

Aurreko ataletan, errekurtsibitatea eta murgilketa erabiliz funtzio asko definitu ditugu. Funtzio horietako batzuk dagoeneko Haskell lengoaian definituta daude (izen desberdin batekin). Dagoeneko Haskell lengoaian definituta dauden funtzioei aurredefinitutako funtzioak deitzen zaie. Atal honetan, zerrendentzat Haskell-en aurredefinituta dauden funtzio batzuk aipatuko ditugu. Gauza bera egiten duten funtzioak defini ditzakegun arren, ondo dator aurredefinitutako funtzioak ezagutzea, horrela ez baitaukagu geuk definitutako funtzioak dituen fitxategia eskura eduki beharrik.

Aipatuko diren funtzio batzuk oinarrizkoa den Prelude moduluan daude definituta. Beste funtzio batzuk, aldiz, Data.List moduluan daude definituta eta funtzio horiek erabiltzeko modulu hori inportatu beharko da.

7.8.1. Prelude oinarrizko modulukoak diren funtzioak

Atal honetan aipatuko diren funtzioak oinarrizkoa den Prelude moduluan daude definituta eta, ondorioz, beti erabilgarri daude.

head :: [t] -> t — Zerrendako lehenengo elementua itzuliko du.

head [4, 6, 8] → 4 head [] → error

tail :: [t] -> [t] — Zerrendako lehenengo elementua kenduz geratzen den zerrenda itzuliko du.

tail [4, 6, 8] → [6, 8] tail [] → error

last :: [t] -> t — Zerrendako azkeneko elementua itzuliko du.

last [4, 6, 8] → 8 last [] → error

init :: [t] -> [t] — Zerrendako azkeneko elementua kenduz geratzen den zerrenda itzuliko du.

init [4, 6, 8] → [4, 6] init [] → error

null :: [t] -> Bool — Zerrenda hutsa al den erabakiko du.

null [4, 6, 8] → False null [] → True

length :: [t] -> Int — Zerrendaren luzera (elementu kopurua) itzuliko du.

length [4, 6, 8] → 3 length [] → 0

sum :: Num t => [t] -> t

Zerrendako elementuen batura itzuliko du. Elementuek zenbakizkoak izan beharko dute.

sum [4, 6, 8] → 18 sum [] → 0

product :: Num t => [t] -> t

Zerrendako elementuen biderkadura itzuliko du. Elementuek zenbakizkoak izan beharko dute.

product [4, 6, 10] → 240 product [] → 1

reverse :: [t] -> [t] Alderantzizko zerrenda itzuliko du.

reverse [4, 6, 8] → [8, 6, 4] reverse [] → []

++ :: [t] -> [t] -> [t]

Bi zerrenda emanda, bi zerrendak elkartzen edo kateatzen ditu zerrenda bakarra osatuz eta ordena mantenduz.

```
[5, 2] ++ [4, 9, 8] → [5, 2, 4, 9, 8]
[] ++ [4, 9, 9, 8] → [4, 9, 9, 8]
[5, 2] ++ [] → [5, 2]
```

Prefixu eran ere erabil daiteke:

```
(++) [5, 2] [4, 9, 8] → [5, 2, 4, 9, 8]
(++) [] [4, 9, 9, 8] → [4, 9, 9, 8]
(++) [5, 2] [] → [5, 2]
```

`elem` :: Eq t => t -> [t] -> Bool

Elementu bat zerrenda batean al dagoen erabakiko du. Elementuentzat berdina izatea zer den definituta egon beharko du.

```
9 `elem` [4, 9, 9, 8] → True
7 `elem` [4, 9, 9, 8] → False
9 `elem` [] → False
```

Prefixu eran ere erabil daiteke:

```
(elem) 9 [4, 9, 9, 8] → True
(elem) 7 [4, 9, 9, 8] → False
(elem) 9 [] → False
```

`notElem` :: Eq t => t -> [t] -> Bool

Elementu bat zerrenda batean ez al dagoen erabakiko du. Elementuentzat berdina izatea zer den definituta egon beharko du.

```
9 `notElem` [4, 9, 9, 8] → False
7 `notElem` [4, 9, 9, 8] → True
```

9 `notElem` [] → True

9 `notElem` [4, 9, 9, 8] eta
not (9 `elem` [4, 9, 9, 8]) baliokideak dira

Prefixu eran ere erabil daiteke:

(notElem) 9 [4, 9, 9, 8] → False
(notElem) 7 [4, 9, 9, 8] → True
(notElem) 9 [] → True

take :: Int -> [t] -> [t]

Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu hartuz osatutako zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda osoa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda hutsa itzuliko da. Zenbakiak Int motakoa izan behar du.

take 2 [4, 6, 8] → [4, 6]
take 5 [4, 6, 8] → [4, 6, 8]
take (-2) [4, 6, 8] → []
take 0 [4, 6, 8] → []

takeWhile :: (t -> Bool) -> [t] -> [t]

Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak False balioa ematen dion zerrendako lehenengo elementura arteko elementuez osatutako zerrenda itzuliko du. Beraz, funtzioak False ematen dion elementu bat aurkitu arte edo zerrenda bukatu arte jarraituko du.

takeWhile (< 3) [1, 2, 3, 4, 1, 2, 3, 4] == [1, 2]
takeWhile (< 9) [1, 2, 3] == [1, 2, 3]
takeWhile (< 0) [1, 2, 3] == []

drop :: Int -> [t] -> [t]

Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu kendu ondoren geratzen den zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa baldin bada, zerrenda hutsa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda osoa itzuliko da. Zenbakiak Int motakoa izan beharko du.

drop 2 [4, 6, 8] → [8]
drop 5 [4, 6, 8] → []
drop (-2) [4, 6, 8] → [4, 6, 8]
drop 0 [4, 6, 8] → [4, 6, 8]

dropWhile :: (t -> Bool) -> [t] -> [t]

Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak False balioa ematen dion zerrendako lehenengo elementura arteko elementuak kenduz osatutako zerrenda itzuliko du. Beraz funtzioak False ematen dion elementu bat aurkitu arte edo zerrenda bukatu arte jarraituko du.

dropWhile (< 3) [1, 2, 3, 4, 1, 2, 3, 4] == [3, 4, 5, 1, 2, 3]
dropWhile (< 9) [1, 2, 3] == []
dropWhile (< 0) [1, 2, 3] == [1, 2, 3]

maximum :: Ord t => [t] -> t

Zerrendako balio handiena itzuliko du. Zerrendako elementuentzat ordenak definituta egon beharko du (txikiagoa izatea, handiagoa, eta abar).

maximum [4, 9, 9, 8] → 9 maximum [] → error

minimum :: Ord t => [t] -> t

Zerrendako balio txikiena itzuliko du. Zerrendako elementuentzat ordenak definituta egon beharko du (txikiagoa izatea, handiagoa, eta abar).

minimum [4, 9, 9, 8] → 4 minimum [] → error

concat :: [[t]] -> [t]

Zerrendez osatutako zerrenda bat emanda, zerrenda denak elkartuz zerrenda bakarra itzuliko du.

concat [[1, 2, 3], [4], [], [5, 6]] → [1, 2, 3, 4, 5, 6]

and :: [Bool] -> Bool

Balio Boolearrez osatutako zerrenda bat emanda, denak True badira, True eta bestela False itzuliko du.

and [4 > 1, True, 5 == 5] → True
and [4 < 1, True, 5 == 5] → False

or :: [Bool] -> Bool

Balio Boolearrez osatutako zerrenda bat emanda, gutxienez bat True bada, True eta bestela, False itzuliko du.

or [4 > 1, True, 5 == 5] → True
or [4 < 1, True, 5 == 5] → True

map :: (t1 -> t2) -> [t1] -> [t2]

Funtzio bat eta zerrenda bat emanda, funtzioa zerrendako elementu bakoitzari aplikatuz osatzen den zerrenda berria itzuliko du.

map (>5) [3, 8, 5, 9] → [False, True, False, True]

map (+2) [3, 8, 5, 9] → [5, 10, 7, 11]

any :: (t -> Bool) -> [t] -> Bool

Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak zerrendako elementuren batentzat True itzultzen badu, orduan any-k ere True itzuliko du eta bestela False itzuliko du.

any (>5) [3, 8, 5, 9] → True
any (>10) [3, 8, 5, 9] → False

all :: (t -> Bool) -> [t] -> Bool

Balio Boolearra itzultzen duen funtzio bat eta zerrenda bat emanda, funtzioak zerrendako elementu denentzat True itzultzen badu, orduan all-ek ere True itzuliko du eta bestela False itzuliko du.

all (>5) [3, 8, 5, 9] → False
all (>2) [3, 8, 5, 9] → True

zip :: [t1] -> [t2] -> [(t1, t2)]

Bi zerrenda emanda, posizio berean dauden elementuekin eratutako bikoteez osatutako zerrenda itzuliko du. Zerrenda bat bestea baino laburragoa bada, bikote-eraketa zerrenda laburrena bukatzean bukatuko da.

zip [1, 2, 3] [4, 5, 6] → [(1, 4), (2, 5), (3, 6)]
zip [1, 2] [4, 5, 6] → [(1, 4), (2, 5)]

Azpimarratzekoa da zip [1, 2, 3] [4, 5, 6] eta
[(x, y) | x <- [1, 2, 3], y <- [4, 5, 6]] zerrenda desberdinak direla, izan ere azken hau honako zerrenda hau da:

[(1, 4), (1, 5), (1, 6), (2, 4), (2, 5), (2, 6), (3, 4), (3, 5), (3, 6)]

zip3 :: [t1] -> [t2] -> [t3] -> [(t1, t2, t3)]

Hiru zerrenda emanda, posizio bereko elementuez eratutako hirukoteen zerrenda itzuliko du.

zip3 [1, 2, 3] [4, 5, 6] [7, 8, 9]→ [(1, 4, 7), (2, 5, 8), (3, 6, 9)]
zip3 [1, 2] [4, 5, 6] [7, 8, 9]→ [(1, 4, 7), (2, 5, 8)]

7.8.2. Data.List modulukoak diren funtzioak

Atal honetan aipatuko diren funtzioak Data.List modulukoak dira eta erabili ahal izateko honako agindu hau ipini beharko da:

import Data.List

Data.List modulua Haskell lengoaian aurredefinituta dagoen modulu bat da eta agindua lerroa ipintzea nahikoa da bertako funtzioak erabiltzeko.

genericLength :: Num i => [t] -> i
Zerrendaren luzera (elementu kopurua) itzuliko du zenbakizko edozein i mota erabiliz.

genericLength [4, 6, 8] → 3 genericLength [] → 0

genericTake :: Integral t1 => t1 -> [t2] -> [t2] Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu hartuz osatutako zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa baldin bada, zerrenda osoa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda hutsa itzuliko da. Zenbakiak Int edo Integer motakoa izan beharko du, hau da, Integral erakoa.

genericTake 2 [4, 6, 8] → [4, 6]
genericTake 5 [4, 6, 8] → [4, 6, 8]
genericTake (-2) [4, 6, 8] → []
genericTake 0 [4, 6, 8] → []

genericDrop :: Integral t1 => t1 -> [t2] -> [t2]
Zenbaki bat eta zerrenda bat emanda, zerrendaren hasieratik zenbakiak adierazten duen adina elementu kendu ondoren geratzen den zerrenda itzuliko du. Zenbakia zerrendaren luzera baino handiagoa bada, zerrenda hutsa itzuliko da. Zenbakia zero edo negatiboa baldin bada, zerrenda osoa itzuliko da. Zenbakiak Int edo Integer motakoa izan behar du, hau da Integral erakoa.

genericDrop 2 [4, 6, 8] → [8]
genericDrop 5 [4, 6, 8] → []
genericDrop (-2) [4, 6, 8] → [4, 6, 8]
genericDrop 0 [4, 6, 8] → [4, 6, 8]

7.8.3. Bereziki String motarentzat Prelude moduluan dauden funtzioak

Preluden, bereziki String motarentzat definituak izan diren funtzio batzuk daude. Funtzio horietako batzuk honako hauek dira:

- **lines**: Karaktere-kate bat emanda, karaktere kate hori osatzen duten lerroez eratutako zerrenda itzuliko du. Lerro-jauzia '\n' espresioaren bidez adierazten da. Funtzio honen mota honako hau da:

```
lines :: String -> [String]
```

Adibidez,

```
lines "aaa\nbbbb\nccc"
```

kasuan erantzuna honako hau da:

```
["aaa","bbbb","ccc"]
```

- **unlines:** Karaktere-katez osatutako zerrenda bat emanda, zerrenda horretako kate bakoitza lerro bat dela kontsideratuz osatzen den katea itzuliko du. Beraz, zerrenda osatzen duten kateak elkartu egingo dira '\n' lerro-jauzia tartekatuz. Funtzio honen mota honako hau da:

```
unlines :: [String] -> String
```

Adibidez,

```
unlines ["aaa","bbbb","ccc"]
```

kasuan erantzuna honako hau izango da:

```
"aaa\nbbbb\nccc\n"
```

- **words**: Karaktere-kate bat emanda, karaktere kate hori osatzen duten hitzez eratutako zerrenda itzuliko du. Zuriuneak eta lerro-jauziak ('\n') dira hitzen mugatzailetzat hartuko direnak. Funtzio honen mota honako hau da:

```
words :: String -> [String]
```

Adibidez,

```
words "aaa bbbb\nccc    dd"
```

kasuan erantzuna honako hau izango da:

```
["aaa","bbbb","ccc","dd"]
```

Adibidean ikus daiteke hitzak zuriune batez baino gehiagoz bereizita egon daitezkeela. Zuriunez osatutako bloke osoak bi hitz bereizteko balio du.

- **unwords**: Karaktere-katez osatutako zerrenda bat emanda, zerrenda horretako kate bakoitza hitz bat dela kontsideratuz osatzen den katea itzuliko du. Beraz, zerrenda osatzen duten kateak elkartu egingo dira zuriunea tartekatuz. Funtzio honen mota honako hau da:

```
unwords :: [String] -> String
```

Adibidez,

```
unwords ["aaa","bbbb","ccc   ","d   d"]
```

kasuan erantzuna honako hau izango da:

```
"aaa bbbb ccc    d   d"
```

Adibidean ikusten da sarrerako zerrendako hitzek zuriuneak baldin badituzte, zuriune horiek mantendu egingo direla.

`"aaa bbbb ccc  d  d"` kateari words aplikatzen badiogu, ez dugu lortuko abiapuntuko zerrenda, beste zerrenda hau lortuko dugu:

```
["aaa","bbbb","ccc","d","d"]
```

Eta hor hitz bat gehiago dago.

7.9. Moten gaineko baldintzak

Aurreko ataletako funtzio batzuk edozein t motarentzat definitu ditugu eta funtzio horien definizioan t mota horretako elementuak berdinak al diren aztertu dugu. Zehazki, 7.4.3 ataleko 'badago' funtzioan eta 7.7.5 ataleko kendu_gb, kendu_lag_ald, kendu_mr_ald, kendu_lag_elk eta kendu_mr_elk funtzioetan hori gertatu da. Funtzio horietan Eq t => baldintza ipini behar izan dugu funtzioaren mota ematean. Baldintza horren bidez honako hau adierazten da: funtzio horiek zentzua izango dute t motarentzat berdintasuna eta desberdintasuna definituta baldin badaude. Oinarrizko motentzat (Bool, Int, Char, Integer, [Int], eta beste batzuk) berdintasuna eta desberdintasuna definituta daude baina geuk mota berri bat definitzen badugu, printzipioz mota berri horrentzat berdintasuna eta desberdintasuna ez daude definituta eta geuk zehaztu beharko dugu nola erabaki behar den mota horretako bi elementu berdinak al diren ala ez.

Aurreko atalean aipatu diren aurredefinitutako funtzio batzuen mota ematean ere era horretako baldintzak agertu dira:

- sum funtzioaren kasuan, **sum** :: **Num t =>** [t] -> t mota daukagu. Beraz, Num t => baldintza ipini da. Horren bidez, t motako zerrenda bateko elementuak batu ahal izateko, t motak zenbakizkoa izan beharko duela adierazten da (Int, Integer, Float, …).
- `elem` funtzioaren kasuan, **`elem`** :: **Eq t =>** t -> [t] -> Bool mota daukagu. Beraz, Eq t => baldintza ipini da. Horren bidez, t motako elementu bat t motako zerrenda batean al dagoen erabaki ahal izateko, t motako elementuentzat berdintasuna eta desberdintasuna definituta egon beharko dutela adierazten da.
- maximum funtzioaren kasuan, **maximum** :: **Ord t =>** [t] -> t mota daukagu. Beraz, Ord t => baldintza ipini da. Horren bidez, t motako zerrenda bateko balio handiena kalkulatu ahal izateko, t motako elementuentzat ordenak (txikiagoa izatea, handiagoa izatea, berdina izatea) definituta egon beharko duela adierazten da.
- genericTake funtzioaren kasuan, **genericTake** :: **Integral t1** => t1 -> [t2] -> [t2] mota daukagu. Beraz, Integeral t1 => baldintza ipini da. Horren bidez, funtzio horri emango zaion lehenengo datuari dagokion t1 motak zenbaki osozkoa (Int, Integer) izan behar duela adierazten da.
- Eta abar.

Baldintzak mota ezezagunekin (t, t1, t2, …) ari garenean erabili beharko dira, aipatutako adibideetan ikusi den moduan.

Moten gainean ezarri daitezkeen baldintza batzuk aipatuko ditugu orain:

- **Eq** t: t motako elementuak berdinak al diren ala ez erabakitzeko erak definituta egon behar duela adierazteko. t mota bateko elementuentzat funtzio bat definitzean t motako elementuak (==) eta (/=) eragileen bidez konparatzen badira, orduan Eq t baldintza ipini beharko da.
- **Ord** t: t motako elementuen artean ordenak (handiagoa, txikiagoa, berdina) definituta egon behar duela adierazteko. t mota bateko elementuentzat funtzio bat definitzean t motako elementuak (<), (<=), (>) eta (>=) eragileen bidez konparatzen badira, orduan Ord t baldintza ipini beharko da.

- **Show** t: t motako elementuak pantailan aurkezteko erak definituta egon behar duela adierazteko. Mota berri bat definitzen denean (adibidez zuhaitz bitarra, pila, multzoa, pertsona, bezeroa, produktua, eta abar), mota horretako elementuak pantailan nola aurkeztuko diren finkatu behar da. Aurkezteko era ez bada finkatzen, Haskell-ek ez du jakingo nola aurkeztu eta errorea sortuko du. Baldintza hau, funtzio berri bat definitzean 'show' eta 'print' funtzioak erabilitzen direnean ipini beharko da. 'show' eta 'print' funtzioak sarrera/irteera-ri buruzko atalean azalduko dira.
- **Read** t: t motako elementuak karaktere-kate gisa irakurri ahal izango direla eta gero karaktere-kate hori t motara bihurtu ahal izango dela adierazteko. Mota berri bat definitzen denean, mota horretako elementuak teklatu bidez karaktere-kate gisa irakurtzeko eta karaktere-kate hori t motara bihurtzeko modu bat zehaztu beharko da. 'read' funtzioa sarrera/irteera-ri buruzko atalean azalduko da.
- **Bounded** t: t motak beheko muga eta goiko muga izan behar dituela adierazteko. Esate baterako, Int mota beheko eta goiko mugak dituen mota bat da.
- **Num** t: t motak zenbakizkoa izan beharko duela adierazteko. Esate baterako, Int, Integer, Float, Double eta Rational (Integer motako elementuekin osatutako zatikiak) zenbakizko motak dira.
- **Integral** t: t motak zenbaki osozkoa izan beharko duela adierazteko, hau da, Int edo Integer.
- **Fractional** t: adibidez t motak Rational izan beharko duela adierazteko.
- **Floating** t: t motak Float edo Double izan behar duela adierazteko.
- **Enum** t: t motak zenbagarria izan behar duela adierazteko.

Mota berri bat definitzean, goian aipatu diren baldintza batzuk bete beharko dituela adierazi dezakegu. Adibidez:

```
data Pila t = Phutsa | Pilaratu (t, Pila t)
        deriving (Eq, Ord, Show, Read)

data Urtaroa = Negua | Udaberria | Uda | Udazkena
        deriving (Eq, Ord, Enum, Show, Read)
```

Bigarren adibide honetan, Eq eta Ord-en bidez urtaroen izenak konpara daitezkeela (Negua Udaberria baino txikiagoa da, Udaberria Uda baino txikiagoa da eta abar) adierazten da; Enum-en bidez zenbatu daitezkeela; Show-ren bidez pantailan hor agertzen diren bezala aurkeztuko direla eta Read-en bidez izen horiek teklatutik ere irakur daitezkeela esaten da.

Funtzio bat definitzean, baldintza bakarra baldin badago, esate baterako Eq, honela ipiniko genuke:

```
f :: Eq t => …
```

Baldintza bat baino gehiago baldin badaude, esate baterako Eq, Show eta Read, orduan honela ipiniko lirateke:

```
f :: (Eq t, Show t, Read t) => …
```

7.10. Tuplak edo n-koteak

Haskell lengoaian tuplak edo n-koteak definitu eta erabiltzeko era aurkeztuko da atal honetan.

7.10.1. Tuplen edo n-koteen definizio formala

Tupla bat hainbat elementuz osatutako bilduma finitua da. Tupla konkretu bat sortzen denetik tupla horren tamaina edo osagai kopurua ezaguna izango da eta tamaina hori ez da aldatuko. Tuplaren osagai kopurua n baldin bada, orduan tuplari n-kote ere deitu diezaiokegu. Tuplak edo n-koteak, nolabaiteko egitura duten eta hainbat osagaitan deskonposatuak izan daitezkeen objektu matematikoak deskribatzeko erabili ohi dira. 'Tupla' hitza gaztelaniazko honako hitz hauen orokorpenetik sortutakoa da: dupla, tripla, cuádrupla, quíntupla, síxtupla, séptupla, óctupla, ..., n-tupla. Ingelesez eta frantsesez ere 'tuple' hitza erabiltzen da eta 'n-tuple' ere bai. Bi osagaiko tuplei bikote deitu ohi zaie, hiru osagaikoei hirukote, lau osagaikoei laukote eta abar. Kasu orokorrera joz, n osagaiko tuplei n-kote deitu ohi zaie.

Ohikoenak bikoteak dira. Esate baterako, (20, 8) zenbaki osoz eratutako bikote bat da. Haskell lengoaian bikote horren mota (Integer, Integer) izango litzateke. Zerrendak mota bereko elementuz osatuta egoten dira, baina tuplen osagaiak mota desberdinetakoak izan daitezke. Adibidez, (20, 'q', 8, True) laukotea (Integer, Char, Integer, Bool) motakoa da. Tupla batean elementuen ordenak garrantzitsua da. Beraz, (20, 'q', 8, True) laukotea eta ('q', 20, True, 8) laukotea desberdinak dira. Laukote horien mota ere desberdina da: (20, 'q', 8, True) laukotearen mota (Integer, Char, Integer, Bool) da eta ('q', 20, True, 8) laukotearen mota (Char, Integer, Bool, Integer) da.

Formalki, tuplak biderketa kartesiarraren bidez sortutako elementuak dira. Haskell lengoaiako (Integer, Bool) mota Integer × Bool biderketa kartesiarraren inplementazioa da. Ondorioz, zenbaki oso batez eta balio boolear batez eratutako bikote guztien multzoak (Integer, Bool) mota osatzen dute. Oro har, Haskell-eko (t1, t2, …, tn) mota t1 × t2 × … × tn biderketa kartesiarrarekin bat dator. Bestalde, Haskell-eko zerrendak itxidura unibertsalaren bidez sortutako elementuak dira. Haskell-eko [Integer] mota Integer* itxidura unibertsalaren inplementazioa da. Gogora dezagun Integer* itxidura unibertsalak honako bildura infnitua adierazten duela:

$$\text{Integer}^* = \text{Integer}^0 \cup \text{Integer}^1 \cup \text{Integer}^2 \cup \ldots$$

Bildura horretan, Integer^0 zerrenda hutsaz osatutako multzoa da, Integer^1 zenbaki oso bakar batez osatutako zerrendez eratutako multzoa da, Integer^2 bi zenbaki osoz eratutako zerrendez osatutako multzoa da eta abar. Oro har, Haskell-eko [t] mota t* itxidura unibertsalaren inplementazioa da. Mota bereko zerrendek luzera desberdina izan dezakete itxidura unibertsalean zero luzerakoak bat luzerakoak, bi luzerakoak eta abar sartzen direlako, baina tuplen kasuan, mota bereko tupla denek luzera bera izango dute biderketa kartesiar batean osagai kopurua finkoa delako. Beraz, (t1, t2, …, tn) motako tupla baten osagai kopurua finkoa da eta bere osagaiak mota desberdinetakoak izan baitezke. Aldiz, [t] motako zerrenda bateko elementu kopurua edozein izan daiteke (0, 1, 2, …) baina elementu denak mota berekoak izan behar dute, t motakoak.

Zerrendak eta tuplak murriztapenik gabe konbinatu daitezke elkarrekin. Adibidez, motako hirukoteak (Integer, [Char], [Integer]) eduki ditzakegu. (15, ['a', 'v', 'b', 'v'], [10, -6, 7, 7, 12]) hirukotea mota horretako elementu bat izango litzateke. (Integer, [Char], [Integer]) mota biderketa kartesiarra eta itxidura unibertsalaren bidez honela adieraziko genuke: Integer × Char* × Integer*. Era berean, [(Bool, Integer, [Char])] motako zerrendak eduki ditzakegu. Mota horretako zerrenda bat honako hau izango litzateke:

[(True, 3, ['a', 'v', 'b', 'v']), (True, 17, ['h', 'h', 'r']), (False, 100, []), (True, 3, ['b', 'v'])]

Zerrenda horretan lau elementu ditugu. Elementu horietako bakoitza (Bool, Integer, [Char]) motako hirukote bat da. [(Bool, Integer, [Char])] mota biderketa kartesiarra eta itxidura unibertsalaren bidez honela adieraziko genuke: (Bool × Integer × Char*)*.

Tuplen arloan, osagai bakarreko tuplak eta tupla hutsa ditugu kasu berezi gisa:

- t motako elementu bakar batez osatutako tuplak t^1 multzokoak dira, baina matematikoki $t^1 = t$ denez, (x) erako tuplak x eran sinplifikatuta adierazten dira gehienetan.
- Zero osagai dituen t motako tupla t^0 multzokoa da. Matematikoki $t^0 = \{()\}$ da, eta () elementua tupla hutsa da. Haskell lengoaian tupla hutsaren mota erabil daiteke eta mota horretako elementu bakarra tupla hutsa da. Haskell-en bai tupla hutsaren mota eta bai tupla hutsa () eran adierazten dira. Funtzio denek emaitza bat itzuli behar dutenez, berez ezer itzuliko ez lukeen funtzio batek zerbait itzuli dezan erabil dezakegu tupla hutsaren mota. Horrelako erabilera ematen zaio hain zuzen ere irteerako funtzioetan. Irteerako funtzioek berez mezu bat aurkeztuko dute pantailan eta ez dute emaitzik itzuliko, baina zerbait itzultzea derrigorrezkoa denez, tupla hutsa itzuli ohi dute.

7.10.2. Tuplak erabiltzen dituzten funtzioen adibideak

Tuplak erabiltzen dituzten funtzio batzuk definituko dira atal honetan.

a) f funtzioak, balio boolear batez eta zenbaki oso batez eratutako bikoteak dituen zerrenda bat emanda, lehenengo osagaitzat True balioa duten bikoteez osatutako zerrenda itzuliko du.

```
f :: [(Bool, Integer)] -> [(Bool, Integer)]
f [] = []
f ((x,y):s)
 | x == True   = (x,y):(f s)
 | x == False  = f s
```

Adibidez, `f [(True,8), (False,6), (True,5)]` kasuan honako zerrenda hau lortuko litzateke:

```
[(True,8),(True,5)]
```

b) g funtzioak, bi zenbaki osoz eratutako bikoteak dituen zerrenda bat emanda, bikoteetako osagaiak ordenatuz lortzen den zerrenda itzuliko du. Beraz, emaitzatzat itzuliko den zerrendako bikote bakoitzean, lehenengo osagaia bigarrena baino txikiagoa edo berdina izango da.

```
g :: [(Integer, Integer)] -> [(Integer, Integer)]
g [] = []
g ((x, y) : s)
 | x > y     = (y, x) : (g s)
 | x <= y    = (x, y) : (g s)
```

Adibidez, `g [(8,9),(7,9),(9,9),(20,-3)]` kasuan honako zerrenda hau lortuko litzateke:

`[(8,9),(7,9),(9,9),(-3,20)]`

c) h1 funtzioak, zenbaki osozko zerrenda bat emanda, zerrenda horretako zenbaki bakoitzeko, hirukote bat duen zerrenda bat itzuliko du. Zenbaki bakoitzari dagokion hirukoteak zenbakia bera, zenbaki horren zatitzaileen zerrenda eta zenbaki horren zatitzaile kopurua izango lituzke osagaitzat. Zerorentzat eta zenbaki negatiboentzat, zatitzaileen zerrendak hutsa izan beharko du eta zatitzaile kopuruak -1. Laguntzaile gisa zatizer funtzioa erabiliko da. Funtzio horrek, zenbaki oso bat emanda, zenbaki horren zatitzaileen zerrenda itzuliko du. Bestalde, zatizer funtzioak zatizer_lag funtzio laguntzailea erabiliko du. Bi funtzio laguntzaile horiek (zatizer eta zatizer_lag) murgilketaren teknikari buruzko atalean definitu dira.

```
h1 :: [Integer] -> [(Integer, [Integer], Integer)]
h1 [] = []
h1 (x:s)
 | x <= 0     = (x, [], -1):(h1 s)
 | x >= 1     = (x, zatizer x, genericLength (zatizer x)):(h1 s)
-----------------------------------------------------------------

-- Funtzio honek, x zenbaki oso bat emanda, [1..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu
-- murgilketa erabiliz.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer:: Integer -> [Integer]
zatizer x = zatizer_lag x 1
-----------------------------------------------------------------

-- Funtzio honek, x eta bm bi zenbaki oso emanda, [bm..x]
-- tartekoak diren x zenbakiaren zatitzaileak itzuliko ditu.
-- x zenbakia 0 edo negatiboa baldin bada, errorea.
-- bm zenbakia 0 edo negatiboa baldin bada, errorea.

zatizer_lag:: Integer -> Integer -> [Integer]
zatizer_lag x bm
```

```
    | x < 0 || x == 0        = error "Zenbakia ez da positiboa."
    | bm <= 0               = error "Beheko muga ez da positiboa."
    | bm > x                = []
    | bm > (x `div` 2)      = [x]
    | x `mod` bm == 0       = bm : (zatizer_lag x (bm + 1))
    | otherwise             = zatizer_lag x (bm + 1)
```

Adibidez, `h1 [4,8,17,-5,0]` kasuan honako zerrenda hau lortuko litzateke:

```
[(4,[1,2,4],3),(8,[1,2,4,8],4),(17,[1,17],2),(-5,[],-1),(0,[],-1)]
```

d) h2 funtzioak, zenbaki oso batez, zenbaki horren zatitzaileen zerrendaz eta zenbaki horren zatitzaile kopuruaz osatutako hirukoteak dituen zerrenda bat emanda, lehenengo osagaitzat zenbaki lehen bat duten hirukoteez osatutako zerrenda itzuliko du. Gogoratu: zenbaki bat lehena izango da bere zatitzaile kopurua 2 baldin bada.

```
h2 :: [(Integer, [Integer], Integer)] -> [(Integer, [Integer], Integer)]
h2 [] = []
h2 ((x,y,z):s)
 | z == 2     = (x,y,z):(h2 s)
 | x /= 2     = h2 s
```

Adibidez,
`h2 [(4,[1,2,4],3),(19,[1,19],2),(8,[1,2,4,8],4),(17,[1,17],2)]`
kasuan honako zerrenda hau lortuko litzateke:

```
[(19,[1,19],2),(17,[1,17],2)]
```

h2 funtzioak ez du egiaztatuko sarrerako zerrenda koherentea al den ala ez. Hau da, ez du aztertuko hirukote bakoitzean bigarren osagaia benetan lehenengo osagaiari dagokion zatitzaile-zerrenda al den eta hirugarren osagaia benetan lehenengo osagaiari dagokion zatitzaile kopurua al den. Adibidez,

```
h2 [(4,[1,2,4],2),(19,[1,19],2),(8,[1,2,12],4),(17,[1,17],10)]
```

kasuan `[(4,[1,2,4],2),(19,[1,19],2)]` lortuko litzateke. Baina hor hirukoteak ez dira koherenteak enuntziatuarekiko. Izan ere, 4ren zatitzaile kopurua ez da 2, 8ren zatitzaileen zerrenda ez [1,2,12] eta 17ren zatitzaile kopurua ez da 10. h2 funtzioak hirugarren osagaitzat 2 balioa duten hirukoteez eratutako zerrenda itzultzen du besterik gabe. Beraz, sarrerako datuak koherenteak ez badira, emaitzatzat h2 funtzioak itzuliko duen zerrendako hirukoteetan gerta daiteke lehenengo osagaia zenbaki lehen bat ez izatea. Adibidera itzuliz, 4 ez da lehena baina `(4,[1,2,4],2)` emaitzan agertzen da eta 17 lehena da baina `(17,[1,17],10)` ez da agertzen emaitzan.

e) h3 funtzioak, edozein motatakoak izan daitezkeen osagaiez eratutako hirukote bat emanda, hirukotearen hirugarren osagaia itzuliko du.

```
h3 :: (t1, t2, t3) -> t3
h3 (x,y,z) = z
```

Adibidez, `h3 (True,'a',20)` kasuan emaitza 20 izango litzateke.

f) h4 funtzioak, zenbaki oso batez, zenbaki horren zatitzaileen zerrendaz eta zenbaki horren zatitzaile kopuruaz osatutako hirukoteak dituen zerrenda bat emanda, lehenengo osagaitzat zenbaki lehen bat duten hirukoteez osatutako zerrenda itzuliko du. Hau da, hirugarren osagaitzat 2 balioa duten hirukoteez osatutako zerrenda itzuliko du. Funtzio honek h2 funtzioak egiten duen gauza bera egiten du, baina beste era batera: h3 funtzioa erabiliz hain zuzen ere.

```
h4 :: [(Integer, [Integer], Integer)] -> [(Integer, [Integer], Integer)]
h4 [] = []
h4 (x:s)
 | h3 x == 2     = x:(h4 s)
 | h3 x /= 2     = h4 s
```

Adibidez,
`h4 [(4,[1,2,4],3),(19,[1,19],2),(8,[1,2,4,8],4),(17,[1,17],2)]`
kasuan honako zerrenda hau itzuliko litzateke:

```
[(19,[1,19],2),(17,[1,17],2)]
```

h4 funtzioak ere ez du egiaztatuko sarrerako zerrenda koherentea al den ala ez. Hau da, ez du aztertuko hirukote bakoitzean bigarren osagaia benetan lehenengo osagaiari dagokion zatitzaile-zerrenda al den eta hirugarren osagaia benetan lehenengo osagaiari dagokion zatitzaile kopurua al den

h2 eta h4 funtzioen arteko desberdintasun nagusia hirukoteen barneko egitura tratatzeko era da. h2 funtzioan, hirukoteen egitura era esplizituan agertzen da (x, y, z) espresioaren bidez, eta hor, x hirukoteko lehenengo osagaia da, y bigarren osagaia eta z hirugarren osagaia. Baina h4 funtzioan, hirukoteen barneko egitura ez da era esplizituan agertzen eta hirukote osoa x aldagaiaren bidez adierazten da. Gero, x hirukote horren hirugarren osagaia lortzeko, h3 funtzioa erabiltzen da.

g) i funtzioak, zenbaki osozko zerrenda bat eta zenbaki osozko bikoteez eratutako zerrenda bat emanda, zenbaki osozko bikoteez eratutako zerrenda bat itzuliko du. Emaitza hori eraikitzeko honako irizpide hau hartuko da kontuan:

- bigarren argumentu gisa emandako zerrendako bikote bakoitzeko, bikoteko lehenengo osagaia lehenengo argumentutzat emandako zerrendan agertzen bada, orduan bikoteko lehenengo osagai horrek eta bikoteko bigarren osagaia gehi bat balioak osatzen duten bikotea agertuko da emaitzako zerrendan; baina bikoteko lehenengo osagaia lehenengo argumentutzat emandako zerrendan ez bada agertzen, orduan bikote horri dagokion bikoterik ez da egongo emaitzako zerrendan.

```
i :: [Integer] -> [(Integer, Integer)] -> [(Integer, Integer)]
```

```
i s [] = []
i s ((x,y):r)
 | x `elem` s      = (x, y + 1):(i s r)
 | x `notElem` s   = i s r
```

Adibidez, `i [6,8,6] [(5,7),(6,20),(8,9),(8,10)]` kasuan honako zerrenda hau itzuliko litzateke:

```
[(6,21),(8,10),(8,11)]
```

7.10.3. Tuplekin zerikusia duten aurredefinitutako funtzioak

Prelude oinarrizko moduluan tuplekin zerikusia duten aurredefinitutako funtzioak badaude. Atal honetan garrantzitsuenak aipatuko dira.

- **fst**: bikote bat emanda, bikoteko lehenengo osagaia itzuliko du. Funtzio horren mota honako hau da:
  ```
  fst :: (t1, t2) -> t1
  ```
  Adibidez, `fst (4,True)` kasuan emaitza 4 da.

- **snd**: bikote bat emanda, bikoteko bigarren osagaia itzuliko du. Funtzio horren mota honako hau da:
  ```
  snd :: (t1, t2) -> t2
  ```
  Adibidez, `snd (4,True)` Adibidez

- **zip**: bi zerrenda emanda, posizio berean dauden elementuekin osatutako bikoteak dituen zerrenda itzuliko du. Datutzat emandako zerrendetako batek besteak baino elementu gutxiago baldin baditu, bikote-eraketa zerrenda laburrena bukatzean amaituko da. Funtzio horren mota honako hau da:

  ```
  zip :: [t1] -> [t2] -> [(t1, t2)]
  ```

  Adibidez, `zip [3, 10, 5, 10] ['b', 'k', 'c', 'e', 'd']` kasuan honako zerrenda hau lortuko da:
  `[(3,'b'),(10,'k'),(5,'c'),(10,'e')]`.

- **unzip**: bikotez osatutako zerrenda bat emanda, bi zerrendez eratutako bikote bat itzuliko du: lehenengo zerrenda bikoteetako lehenengo osagaiez eratuta egongo da eta bigarren zerrenda bikoteetako bigarren osagaiez eratuta egongo da. Funtzio horren mota honako hau da:

  ```
  unzip :: [(t1, t2)] -> ([t1], [t2])
  ```

  Adibidez, `unzip [(3,True),(10,True),(5,False),(10,True)]` kasuan honako bikote hau lortuko da:
  `([3,10,5,10],[True,True,False,True])`

Oraintxe ikusi dugu zip eta unzip funtzioak bikoteekin aritzen direla. Hirukoteekin aritzeko, laukoteekin aritzeko eta horrela zazpikoteetaraino, zip3, zip4, …, zip7 eta unzip3, unzip4, …, unzip7 aurredefinitutako funtzioak ditugu.

zip4, …, zip7 funtzioek eta unzip4, …, unzip7 funtzioek Data.List modulua behar dute.

Aurredefinitutako funtzioak eskura edukitzea oso ondo etortzen da askotan baina hala ere, garbi eduki behar dugu aurredefinitutako funtzio horietako asko nahiko era errazean geuk ere definitu ditzakegula. Adibidez, *gure_zip* funtzioak aurredefinitutako zip funtzioak egiten duen gauza bera egiten du:

```
gure_zip :: Eq t2 => [t1] -> [t2] -> [(t1, t2)]
gure_zip [] r = []
gure_zip (x:s) r
 | r == []      = []
 | r /= []      = (x, head r): (gure_zip s (tail r))
```

Definizio horretan, Eq t2 => baldintza ipini beharra dago. Izan ere, [t2] motakoa den r zerrenda berdintza batean agertzen da eta konparazio hori egin ahal izateko, t2 motarentzat berdintasunak eta desberdintasunak definituta egon behar dute.

Kalkulu bera egiten duten funtzioak definitzen baditugu, gerta daiteke motei buruzko baldintzak desberdinak izatea. Adibidez, *gure_zip2* funtzioak *gure_zip* funtzioak egiten duen kalkulu bera egiten du baina funtzioa era desberdinean definitu da. gure_zip2 funtzioaren kasuan, lehenengo zerrenda [t1] motakoa da berdintza batean agertzen da. Ondorioz, Eq t1 => baldintza ipini behar da mota ematean.

```
gure_zip2 :: (Eq t1,Eq t2) => [t1] -> [t2] -> [(t1, t2)]
gure_zip2 s r
 | s == []          = []
 | r == []      = []
 | otherwise    = (head s, head r): (gure_zip2 (tail s) (tail r))
```

7.11. Zerrenda-eraketa

Haskell-en erabili daitekeen **zerrenda-eraketa**rako teknika landuko dugu atal honetan. Teknika hori era isolatuan edo gai honetako aurreko ataletan landutako errekurtsibitatearekin nahasian erabil daiteke.

7.11.1. Notazioa

Haskell-eko zerrenda-eraketaren teknika matematikan multzoak definitzeko erabili ohi den teknikaren antzekoa da. Teknika matematiko hori dagoeneko irakasgai honetako 2. gaian erabili dugu lengoaiak hitzez osatutako multzo gisa definitzeko.

Matematikan multzoak honako era honetan definitu edo adierazi ohi dira askotan:

$$\{ x \mid x \in D \wedge P(x)\}$$

Espresio horren esanahia honako hau da: "D definizio-eremukoak diren eta P propietatea (edo baldintza) betetzen duten x elementuez osatutako multzoa".

Hona hemen adibide bat:

$$\{ x \mid x \in N \wedge x \bmod 10 = 0\}$$

Hor, zenbaki arrunten eremukoak diren eta 10 zenbakiaren anizkoitzak izatearen propietatea (edo baldintza) betetzen duten zenbakiez osatutako multzoa daukagu.

Hona hemen beste adibide bat:

$$\{ x \mid 20 \leq x \leq 100 \wedge \neg\exists y(2 \leq y \leq x - 1 \wedge x \bmod y = 0)\}$$

Hor 20 eta 100 zenbakiek mugatzen duten tartekoak edo eremukoak diren eta lehena izatearen propietatea (edo baldintza) betetzen duten zenbakiez osatutako multzoa daukagu. Beraz, $20 \leq x \leq 100$ eremua izango litzateke eta $\neg\exists y(2 \leq y \leq x - 1 \wedge x \bmod y = 0)$ propietatea izango litzateke.

2. gaian landutako multzoen definizioetara itzuliz, egitura bera dutela ikus dezakegu:

$$\{ w \mid w \in A^* \wedge |w| \bmod 2 = 0\}$$

Hor eremua A^* izango litzateke (hau da, A alfabetoaren gainean definitutako lengoaia unibertsala) eta propietatea luzera bikoitia edukitzea izango litzateke. Beraz, A alfabetoaren gainean definitutakoak diren eta luzera bikoitia duten hitzez osatutako lengoaia definitu da.

Multzoak definitzeko egitura hori jarraituz, zerrendak definitzeko aukera eskaintzen du Haskell-ek. Zerrendak definitzeko honako egitura hau izango genuke:

[x | x <- D, P(x)]

Beraz, matematikako { eta } sinboloen ordez [eta] sinboloak erabili behar dira, $\in$ sinboloaren ordez <- erabili behar da eta $\wedge$ sinboloaren ordez koma erabili behar da.

7.11.2. Oinarrizko adibideak

Jarraian, zerrenda-eraketaren teknika darabilten oinarrizko adibide batzuk azalduko dira.

1. adibidea:
Osoa den n zenbakia emanda, 0tik n-ra arteko zenbaki denak dituen zerrenda itzuliko du *zenbakiak* izeneko honako funtzio honek.

```
zenbakiak:: Int -> [Int]

zenbakiak n = [ x | x <- [0..n]]
```

Adibidez,
```
zenbakiak 5
```
kasuan, [0, 1, 2, 3, 4, 5] zerrenda lortuko litzateke.

Gauza bera egiteko beste aukera bat:

```
zenbakiak2:: Int -> [Int]

zenbakiak2 n = [0..n]
```

2. adibidea:
Osoa den n zenbakia emanda, 0tik n-ra arteko zenbaki bikoiti denak dituen zerrenda itzuliko du *bikoitiak* izeneko honako funtzio honek.

```
bikoitiak:: Int -> [Int]

bikoitiak n = [ x | x <- [0..n], x `mod` 2 == 0]
```

Adibidez,
```
bikoitiak 5
```
kasuan, emaitza [0, 2, 4] izango litzateke.

3. adibidea:
Osoa den n zenbakia emanda, [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikote denez eratutako zerrenda itzuliko du *bikoteak* izeneko honako funtzio honek.

```
bikoteak:: Int -> [(Int, Int)]

bikoteak n = [ (x, y) | x <- [0..n], y <- [0..n]]
```

```
```

Adibidez,

bikoteak 3

kasuan, honako zerrenda hau lortuko litzateke:
[(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 0), (2, 1), (2, 2), (2, 3), (3, 0), (3, 1), (3, 2), (3, 3)]

4. adibidea:

Negatiboak ez diren zenbaki osoz osatutako (x, y) erako bikote denez eratutako zerrenda lortzeko *bikoteak_infinitua* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_infinitua:: [(Integer, Integer)]

bikoteak_infinitua = [ (x, y) | x <- [0..], y <- [0..]]
```

[0..] zerrendaren bidez negatiboak ez diren zenbaki osoen {0, 1, 2, 3, …} multzo infinitua adieraz dezakegu.

bikoteak_infinitua funtzioari deitzen badiogu, honako zerrenda infinitu hau itzuliko du:
[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 9), (0, 10), (0, 11), …]

Zerrenda horretan (1, 0), (1, 1), (1, 2), …(2, 0), (2, 1), (2, 2), (2, 3), …erako bikoteak ere agertu beharko lukete baina bikoteak_infinitua funtzioak ez ditu inoiz sortuko. Funtzio horrek lehenengo osagaitzat 0 balioa duten bikoteak sortuz joango da amaierarik gabe. 2. gaian, N × N multzoaren zenbagarritasuna aztertu genuenean, N × N multzoko bikoteak ordenatzeko era egokia honako hau zela esan genuen:

[(0, 0), (0, 1), (1, 0), (0, 2), (1, 1), (2, 0), (0, 3), (1, 2), (2, 1), (3, 0), …]

Baina Haskell-ek ez du sortuko automatikoki orden egoki hori. Momentuz honela lagako dugu problema hau baina aurrerago ariketa gisa planteatuko dugu.

5. adibidea:

Osoa den n zenbaki bat emanda, *bikoteak_finitua* izeneko funtzioak bikoteak_infinitua funtzioa sortuz joango den zerrenda infinituko lehenengo n bikoteez osatutako zerrenda itzuliko du:

```
bikoteak_finitua:: Integer -> [(Integer, Integer)]

bikoteak_finitua n = genericTake n bikoteak_infinitua
```

Definizio horretan, aurredefinitutako *genericTake* funtzioa erabili da. Osoa den n zenbaki bat eta zerrenda bat emanda, *genericTake* funtzioak zerrendako lehenengo n elementuez osatutako zerrenda itzuliko du.

Adibidez,

bikoteak_finitua 5

kasuan, honako zerrenda finitua lortuko litzateke:

[(0, 0), (0, 1), (0, 2), (0, 3), (0, 4)]

6. adibidea:

Osoa den n zenbakia emanda, lehenengo osagaia bigarrena baino txikiagoa duten [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikote denez eratutako zerrenda lortzeko *bikoteak_hand* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_hand:: Integer -> [(Integer, Integer)]

bikoteak_hand n = [ (x, y) | x <- [0..n], y <- [0..n], x < y]
```

Adibidez,

bikoteak_hand 3

kasuan, [(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)] zerrenda lortuko litzateke.

7. adibidea:

Osoa den n zenbakia emanda, [0..n] tartekoak diren zenbakiz osatutako (x, y) erako bikoteei dagozkien baturez eratutako zerrenda lortzeko *bikoteak_batu* izeneko honako funtzio hau defini dezakegu.

```
bikoteak_batu:: Int -> [Int]

bikoteak_batu n = [ x + y | x <- [0..n], y <- [0..n]]
```

Adibidez,

bikoteak_batu 3

kasuan, [0 + 0, 0 + 1, 0 + 2, 0 + 3, 1 + 0, 1 + 1, 1 + 2, 1 + 3, 2 + 0, 2 + 1, 2 + 2, 2 + 3, 3 + 0, 3 + 1, 3 + 2, 3 + 3] zerrenda itzuliko luke, baina eragiketak eginda

[0, 1, 2, 3, 1, 2, 3, 4, 2, 3, 4, 5, 3, 4, 5, 6]

8. adibidea:

Osoa den n zenbakia emanda, [], [1], [1, 2], [1, 2, 3], …, [1, 2, 3, …, n] zerrendez osatutako zerrenda lortzeko *zerrendak* izeneko honako funtzio hau defini dezakegu.

```
zerrendak:: Int -> [[Int]]

zerrendak n = [ [1..x] | x <- [0..n]]
```

Adibidez,

zerrendak 3

kasuan, [[], [1], [1, 2], [1, 2, 3]] zerrenda lortuko litzateke.

9. adibidea:

Osoa den n zenbakia emanda, n-ren zatitzaile denez osatutako zerrenda kalkulatuko duen *zatizer_ze* funtzioa honela defini daiteke zerrenda-eraketaren teknika erabiliz.

```
zatizer_ze:: Integer -> [Integer]

zatizer_ze n = [ x | x <- [1..n], n `mod` x == 0]
```

Adibidez,
zatizer_ze 12
kasuan, [1, 2, 3, 4, 6, 12] zerrenda lortuko litzateke.
Eta
zatizer_ze 7
kasuan [1, 7] zerrenda lortuko litzateke.

Kasu berezi gisa, n-ren balioa 0 baldin bada edo negatiboa baldin bada, [1..n] tartea hutsa izango da eta ondorioz zatizer_ze funtzioak zerrenda hutsa itzuliko du.

Beraz,
zatizer_ze (-5)
kasuan, [] zerrenda lortuko litzateke.

Bestalde, n-ren balioa 0 edo negatiboa denean zerrenda hutsa itzuli beharrean errore-mezu bat aurkeztea nahi izanez gero, honako funtzio hau defini genezake:

```
zatizer2_ze:: Integer -> [Integer]

zatizer2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = [ x | x <- [1..n], n `mod` x == 0]
```

Eraginkorragoa den funtzio bat ere defini genezake [2..(n `div` 2)] tartea bakarrik aztertuz:

```
zatizer3_ze:: Integer -> [Integer]

zatizer3_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | n == 1        = [1]
        | otherwise     = [1] ++ [ x | x <- [2..( n `div` 2)], n `mod` x == 0] ++ [n]
```

Hirugarren bertsio honi jarraituz, 2 edo handiagoa den zenbaki bat daukagun bakoitzean, 1 eta n zuzenean sartuko dira zatitzaileen zerrendan eta gero [2..(n `div` 2)] tartean bakarrik bilatu beharko da.

zatizer_ze eta zatizer3_ze funtzioen arteko desberdintasuna nabaria da 1000000 bezalako zenbaki handientzat.

10. adibidea:

Osoa den n zenbakia emanda, n lehena al den erabakiko duen *lehena_ze* funtzioa honela defini daiteke zerrenda-eraketaren teknika erabiliz.

```
lehena_ze:: Integer -> Bool

lehena_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | n == 1        = False
        | otherwise     = (length [ x | x <- [2..( n `div` 2)], n `mod` x == 0]) == 0
```

Adibidez,
lehena_ze 12
kasuan, erantzuna False izango da.
Eta
lehena_ze 7
kasuan, erantzuna True izango da.

11. adibidea:

Osoa den n zenbakia emanda, n zenbakiaren faktoriala itzuliko duen *fakt_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika eta aurredefinitutako funtzioak erabiliz.

```
fakt_ze:: Integer -> Integer

fakt_ze n = product [1..n]
```

Funtzio horrek 1 balioa itzuliko du n negatiboa denean. Izan ere, n-ren balioa 1 baino txikiagoa denean, [1..n] tartea hutsa izango da eta aurredefinitutako product funtzioak 1 itzuliko du. Zenbaki negatiboentzat emaitza moduan 1 itzuli beharrean errore-mezua aurkeztea nahi izanez gero, honako funtzio hau defini dezakegu:

```
fakt2_ze:: Integer -> Integer

fakt2_ze n
        | n <= (-1)     = error "Zenbakia negatiboa da"
        | otherwise     = product [1..n]
```

Adibidez,
fakt_ze 4
eta
fakt2_ze 4
kasuetan, 24 balioa lortuko da emaitza gisa.

12. adibidea:

Osoa den n zenbakia emanda, n zenbakia betea al den erabakiko duen *betea_ze* funtzioa definituko dugu orain zerrenda-eraketaren teknika eta aurredefinitutako funtzioak erabiliz.
Positiboa den n zenbaki bat betea da bere zatitzaileen batura (n bera kontuan hartzeke) n denean.

```
betea_ze:: Integer -> Bool

betea_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..n - 1], n `mod` x == 0]) == n
```

betea_ze 6 deiak True itzuliko du 6 = 1 + 2 + 3 baita.
betea_ze 28 deiak True itzuliko du 28 = 1 + 2 + 4 + 7 + 14 betezen baita.
betea_ze 12 deiak False itzuliko du 12 ≠ 1 + 2 + 3 + 4 + 6 baita.

28ren ondoren hurrengo zenbaki betea 496 da.

Zatitzaileak [1..(n `div` 2)] tartean bakarrik bilatuz funtzioaren eraginkortasuna hobetu daiteke.

```
betea2_ze:: Integer -> Bool

betea2_ze n
        | n <= 0        = error "Zenbakia ez da positiboa"
        | otherwise     = (sum [ x | x <- [1..(n `div` 2)], n `mod` x == 0]) == n
```

13. adibidea:

Osoa den n zenbakia emanda, n zenbakiaren faktoreen zerrenda itzuliko duen *faktoreak_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz.

Gogoratu n-ren faktoreak n-ren zatitzaile lehenak direla:

```
faktoreak_ze:: Integer -> [Integer]

faktoreak_ze n
        | n <= 1        = error "Zenbakia 2 baino txikiagoa da"
        | otherwise     = [ x | x <- [2..n], n `mod` x == 0, lehena_ze x]
```

faktoreak_ze 6 deiak [2, 3] itzuliko du.
faktoreak_ze 12 deiak [2, 3] itzuliko du.
faktoreak_ze 8 deiak [2] itzuliko du.
faktoreak_ze 7 deiak [7] itzuliko du..

Funtzio hori hobetzeko, alde batetik n lehena al den aztertu daiteke eta n lehena ez denean zatitzaile lehenen bilaketa [2..(n `div` 2)] tartera muga dezakegu.

```
faktoreak2_ze:: Integer -> [Integer]

faktoreak2_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = [ x | x <- [2..(n `div` 2)], n `mod` x == 0, lehena_ze x]
```

14. adibidea:

Osoa den n zenbakia emanda, n zenbakia faktore lehenetan deskonposatu eta deskonposaketa horri dagokion zerrenda itzuliko duen *desk_ze* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
desk_ze:: Integer -> [Integer]

desk_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = qs((faktoreak2_ze n) ++
                              (desk_ze (n `div` (product (faktoreak2_ze n)))))
```

Hor qs funtzioa zenbaki osozko zerrenda bat ordenatzen duen funtzioa da. qs funtzioaren definizioa hurrengo atalean dago (Adibide gehiago: Murgilketa eta zerrenda-eraketa). Bestalde, faktoreak2_ze funtzioa 13. adibidean definitu da.

desk_ze 6 deiak [2, 3] itzuliko du.
desk_ze 12 deiak [2, 2, 3] itzuliko du.
desk_ze 8 deiak [2, 2, 2] itzuliko du.
desk_ze 7 deiak [7] itzuliko du.

Definizio batean behin baino gehiagotan kalkulatzen den zati bat agertzen denean, **where** erabiliz izen bat eman diezaiokegu.

```
desk2_ze:: Integer -> [Integer]

desk2_ze n
        | n <= 1          = error "Zenbakia 2 baino txikiagoa da"
        | lehena_ze n     = [n]
        | otherwise       = qs(w ++ (desk2_ze (n `div` (product w))))
                              where w = faktoreak2_ze n
```

15. adibidea:

2tik hasita, zenbaki denen faktoreen zerrendak itzuliko dituen *denen_faktoreak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
denen_faktoreak :: [[Integer]]

denen_faktoreak = [faktoreak2_ze y | y <- [2..]]
```

denen_faktoreak funtzioa honako zerrenda infinitua aurkeztuz joango litzateke:
[[2], [3], [2], [5], [2, 3], [7], [2], [3], [2, 5], [11], [2, 3], [13], [2, 7], [3, 5], [2], …

16. adibidea:

Osoa den n zenbakia emanda, 2tik hasita n + 1 zenbakira arteko zenbaki denen faktoreen zerrendak itzuliko dituen *faktoreak_finitua* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
faktoreak_finitua :: Integer -> [[Integer]]

faktoreak_finitua n = [faktoreak2_ze x | x <- [2..(n + 1)]]
```

faktoreak_finitua 5 deiak honako zerrenda finitu hau itzuliko luke:
[[2], [3], [2], [5], [2, 3]]
Zerrenda horretan, 2, 3, 4, 5 eta 6ren faktoreen zerrendak ditugu.

Definizio horri jarraituz, n-ren balioa 0 edo txikiagoa denean zerrenda hutsa itzuliko da. Beste aukera bat n negatiboa denean errore-mezua aurkeztea izango litzateke. Gainera genericTake eta denen_faktoreak funtzioak ere erabil ditzakegu:

```
faktoreak_finitua2 :: Integer -> [[Integer]]

faktoreak_finitua2 n
        | n < 0         = error "Negatiboa"
        | otherwise     = genericTake n denen_faktoreak
```

17. adibidea:

2tik hasita eta amaierarik gabe, zenbaki bakoitza eta zenbakiari dagokion faktore-zerrendaz eratutako bikoteak dituen zerrenda aurkeztuz joango den *faktorizatuak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
faktorizatuak :: [(Integer,[Integer])]

faktorizatuak = zip [2..] denen_faktoreak
```

Faktorizatuak funtzioa honako zerrenda infinitua aurkeztuz joango litzateke:
[(2, [2]), (3, [3]), (4, [2]), (5, [5]), (6, [2, 3]), (7, [7]), (8, [2]), (9, [3]), (10, [2, 5]), (11, [11]), (12, [2, 3]), (13, [13]), (14, [2, 7]), (15, [3, 5]), (16, [2]), …

18. adibidea:

Lehenengo osagai gisa lehena ez den eta faktore bakarra duen zenbakia duten *faktorizatuak* zerrendako bikoteez osatutako zerrenda itzuliko duen *faktore_bakarrekoak* funtzioa honela defini dezakegu:

```
faktore_bakarrekoak :: [(Integer,[Integer])]

faktore_bakarrekoak = [(x,y) | (x,y) <- faktorizatuak, not (lehena_ze x), length y == 1]
```

faktore_bakarrekoak funtzioa honako zerrenda infinitu hau aurkeztuz joango litzateke:
[(4, [2]), (8, [2]), (9, [3]), (16, [2]), (25, [5]), (27, [3]), …

faktore_bakarrekoak funtzioa definitzeko beste aukera bat honako hau da:

```
faktore_bakarrekoak2 :: [(Integer,[Integer])]

faktore_bakarrekoak2 = [(x,y:s) | (x,y:s) <- faktorizatuak, not (lehena_ze x), s == [ ] ]
```

faktore_bakarrekoak funtzioa definitzeko hirugarren aukera:

```
faktore_bakarrekoak3 :: [(Integer,[Integer])]

faktore_bakarrekoak3 = [ (x,[y]) | (x,[y]) <- faktorizatuak, not (lehena_ze x)]
```

Bigarren osagaia elementu bakarreko zerrenda izatea nahi dugunez eta gainera, lehenengo osagaia lehena ez izateagatik lehenengo osagai hori eta dagokion faktore-zerrendako elementu bakarrak berdinak ezin dutenez izan, honako laugarren aukera hau ere eman dezakegu:

```
faktore_bakarrekoak4 :: [(Integer,[Integer])]

faktore_bakarrekoak4 = [ (x, [y]) | (x, [y]) <- faktorizatuak, x /= y]
```

19. adibidea:

Lehenak ez diren eta faktore bakarra duten zenbakiz osatutako zerrenda infinitua aurkeztuz joango den *erabateko_berredurak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_berredurak :: [Integer]

erabateko_berredurak = [ x | (x,y) <- faktore_bakarrekoak]
```

erabateko_berredurak funtzioa honako zerrenda aurkeztuz joango litzateke:
[4, 8, 9, 16, 25, 27, …

20. adibidea:

Osoa den n zenbaki bat emanda, lehenak ez diren eta faktore bakarra duten lehenengo n zenbakiz osatutako zerrenda itzuliko duen *erabateko_lehenengoak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_lehenengoak :: Integer -> [Integer]

erabateko_lehenengoak n

  | n < 0                = error "Balio negatiboa"
  | otherwise            = genericTake n erabateko_berredurak
```

erabateko_lehenengoak 7 deiak honako zerrenda hau itzuliko luke:
[4, 8, 9, 16, 25, 27, 32]

21. adibidea:

Osoak diren n eta m bi zenbaki emanda, m baino handiagoak izanda, lehenak ez diren eta faktore bakarra duten lehenengo n zenbakiz osatutako zerrenda itzuliko duen *erabateko_handiagoak* funtzioa honela defini dezakegu zerrenda-eraketaren teknika erabiliz:

```
erabateko_handiagoak :: Integer -> Integer -> [Integer]

erabateko_handiagoak n m

  | (n < 0) || (m < 0)      = error "Balio negatiboa"
  | otherwise               = genericTake n [ y | y <- erabateko_berredurak, y > m]
```

erabateko_handiagoak 5 12 deiak honako zerrenda hau itzuliko luke:
[16, 25, 27, 32, 64]

7.12. Adibide gehiago: Murgilketa eta Zerrenda-eraketa

Atal honetan programazioan ezagunak diren algoritmoak Haskell-ez emango dira. Aurkeztutako soluzioetan murgilketaren teknika eta zerrenda-eraketaren teknika erabiliko dira.

7.12.1. Ordenatze azkarra (quick sort) zerrenda-eraketa erabiliz

Zerrenda-eraketaren teknikaren abantailak erakusteko, *ordenatze azkarra* edo *quick sort* metodoari jarraituz zerrendak ordenatzen dituen funtzioa garatuko dugu.

Ordenatze azkarraren metodoan x deituko diogun zerrendako lehenengo elemetua hartu, eta zerrendako beste elementuak kontsideratuz, ezkerrean x baino txikiagoak edo berdinak diren elementuez osatutako zerrenda, erdian [x] zerrenda eta eskuinaldean x baino handiagoak diren elementuez osatutako zerrenda ipiniko dira. Urrats horren ondoren x ordenatuta dago eta ezkerreko eta eskuineko zerrendak ordenatzea falta da. Bi zerrenda horiek metodo bera erabiliz ordenatuko dira (errekurtsiboki). Ordenatze-metodo hau "zatitu eta garaile izango zara" teknikaren kasu bat da. Jarraian ordenatze azkarra edo quick sort metodoa erabiliz [5, 8, 3, 2, 9, 7] zerrenda nola ordenatuko litzatekeen azalduko da

- Lehenengo elementua hartu, txikiagoak edo berdinak zerrenda batean ipini eta handiagoak direnak beste zerrenda batean ipini:

Hasierako zerrenda
[5, 8, 3, 2, 9, 7]

berdinak edo txikiagoak direnak	Lehenengo elementua	handiagoak direnak
[3, 2]	++ [5] ++	[8, 9, 7]

- Orain [3, 2] ordenatu behar da:

Hasierako zerrenda
[3, 2]

berdinak edo txikiagoak	Lehenengo elementua	handiagoak direnak

direnak		
[2]	++ [3] ++	[]
Elementu bakarra duenez ordenatuta dago		Hutsa denez ordenatuta dago

Beraz, [3, 2] ordenatuta geratu da eta [2] ++ [3] ++ [] zerrenda lortu da, hau da, [2, 3].

- Orain [8, 9, 7] ordenatu behar da:

Hasierako zerrenda
[8, 9, 7]

berdinak edo txikiagoak direnak	Lehenengo elementua	handiagoak direnak
[7]	++ [8] ++	[9]
Elementu bakarra duenez ordenatuta dago		Elementu bakarra duenez ordenatuta dago

[8, 9, 7] ordenatzea bukatu da eta [7] ++ [8] ++ [9] zerrenda eraiki da, hau da, [7, 8, 9].

- Bukaerako emaitza [2, 3] ++ [5] ++ [7, 8, 9] da eta ++ eragiketari dagozkion kalkuluak eginez [2, 3, 5, 7, 8, 9] geratzen da.

Zerrenda bat ordenatzeko era hau inplementatzen duen funtzioa honela definituko genuke:

```
qs :: [Integer] -> [Integer]

qs [ ] = [ ]
qs (x:s)
        | s == [ ]          = [x]
        | otherwise         = (qs [y | y <- s, y <= x]) ++ [x] ++ (qs [y | y <- s, y > x])
```

Adibidez, qs [5, 8, 3, 2, 9, 7] deiak [2, 3, 5, 7, 8, 9] zerrenda itzuliko du.

Definizio horretan erakutsi da zerrenda hutsa eta elementu bakarreko zerrendak oinarrizko kasutzat har daitezkeela eta horrela dei errekurtsibo gutxiago egin beharko dira, baina berez oinarrizko kasutzat zerrenda hutsa hartzearekin nahikoa da. Hori dela eta, honako definizio hau ere zuzena da:

```
qs2 :: [Integer] -> [Integer]
```

```
qs2 [ ] = [ ]
qs2 (x:s) = (qs2 [y | y <- s, y <= x]) ++ [x] ++ (qs2 [y | y <- s, y > x])
```

7.12.2. Ordenatze azkarra (quick sort) murgilketa erabiliz eraginkortasuna lortzeko (bukaerako errekurtsibitatea)

Ordenatze azkarra edo quick sort metodoa erabiliz zenbaki osozko zerrenda bat ordenatzen duen qs funtzioa definitu da aurreko atalean:

```
qs :: [Integer] -> [Integer]

qs [ ] = [ ]
qs (x:s)
        | s == [ ]          = [x]
        | otherwise         = (qs [y | y <- s, y <= x]) ++ [x] ++ (qs [y | y <- s, y > x])
```

Oinarrizkoa ez den kasuan, hau da, hirugarren kasuan, funtzio horrek bi dei errekurtsibo sortzen ditu eta, ondorioz, adar ugariko zuhaitz bitarra eratuko da funtzioa exekutatzean. Bukaerako errekurtsibitatea azaltzean esan genuen adar ugari eratzen dituzten exekuzioak ez direla eraginkorrak izan ohi. Kasu horietan, murgilketaren teknika erabiliz, adar bakarra sortuko duen eta gainera bukaerako errekurtsibitatea duen funtzioa definitu ahal izango da. Bukaerako errekurtsibitateari esker, hostoa sortutakoan emaitza ere kalkulatuta egongo da.

Orain murgilketaren teknika erabiliz, ordenatze azkarraren metodoa inplementatzen duen qs_lag beste funtzio bat definituko dugu. Funtzio horren exekuzioek bukaerako errekurtsibitatea duen adar bakar bat eratuko dute.

Ordenatu beharreko azpizerrendak gordez joateko, pilaren kontzeptuan oinarrituko gara, nahiz eta gero pila hori zenbaki osozko zerrendez osatutako zerrenda bezala definitu. Gainera, beste parametro bat ere ipini beharko dugu ordenatutako azpizerrendak gordez joateko eta bukaerako zerrenda eraikiz joateko.

- [5, 8, 3, 2, 9, 7] zerrenda ordenatzeko, pila zerrenda horrekin hasieratuko genuke:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua ([] balioarekin hasieratuko da)
[5, 8, 3, 2, 9, 7]	[]

- Jarraian zerrenda hiru zatitan banatuko da: lehenengo elementua (5) erdian geratuko da, 5 baino handiagoak azpian eta 5 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko	Eraikitzen ari garen zerrenda ordenatua

zerrenden pila

[3, 2]
[5]
[8, 9, 7]

[]

Beraz, orain hiru zerrenda horiek ordenatu behar ditugu.

- Gailurrean dagoen zerrenda hartu eta hirutan banatuko dugu: lehenengo elementua (3) erdian geratuko da, 3 baino handiagoak azpian eta 3 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko zerrenden pila

Eraikitzen ari garen zerrenda ordenatua

[2]
[3]
[]
[5]
[8, 9, 7]

[]

Orain bost zerrenda horiek ordenatu behar dira.

- Gailurrean dagoena hartu behar da eta elementu bakarra duenez, ordenatuta dago. Beraz, emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila

Eraikitzen ari garen zerrenda ordenatua

[3]
[]
[5]
[8, 9, 7]

[] ++ [2]

Orain lau zerrenda ditugu ordenatzeko.

- Gailurrean dagoena hartu eta elementu bakarra izateagatik badakigu ordenatuta dagoela eta, ondorioz, emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[]	
[5]	
[8, 9, 7]	[] ++ [2] ++ [3]

Ordenatu beharreko hiru zerrenda geratzen zaizkigu.

- Gailurrean dagoena hutsa da eta hutsa denez ordenatuta dago eta emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[5]	
[8, 9, 7]	[] ++ [2] ++ [3] ++ []

Bi zerrenda geratzen zaizkigu ordenatzeko.

- Gailurrean dagoena hartu eta elementu bakarra duenez, badakigu ordenatuta dagoela eta, zuzenean, emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[8, 9, 7]	[] ++ [2] ++ [3] ++ [] ++ [5]

Orain ordenatu beharreko zerrenda bakarra geratzen zaigu pilan.

- Zerrenda bakar hori hartu eta hirutan banatuko dugu: lehenengo elementua (8) erdian geratuko da, 8 baino handiagoak azpian eta 8 baino txikiagoak edo berdinak direnak gainean.

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua

[7]
[8]
[9]

[] ++ [2] ++ [3] ++ [] ++ [5]

- Berriro hiru zerrenda ditugu pilan. Gailurrean dagoena hartu eta, elementu bakarrekoa denez, ordenatuta dago eta, zuzenean, emaitza eraikitzen ari garen parametroan gordeko dugu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[8] [9]	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7]

Orain pilan bi zerrenda daude ordenatzeko.

- Gailurrekoa ordenatuta dago elementu bakarra izateagatik eta, zuzenean, emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
[9]	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8]

Orain pilan zerrenda bakarra geratzen da.

- Zerrenda horrek elementu bakarra du eta ordenatuta dagoenez emaitza eraikitzen ari garen parametroan gorde dezakegu elkarketaren bidez:

Ordenatu beharreko zerrenden pila	Eraikitzen ari garen zerrenda ordenatua
	[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8] ++ [9]

- Orain pila hutsik dagoenez, [5, 8, 3, 2, 9, 7] ordenatu dugu eta emaitza honako hau da

[] ++ [2] ++ [3] ++ [] ++ [5] ++ [7] ++ [8] ++ [9]

Eragiketak burutu ondoren zerrenda hau geratzen da:

[2, 3, 5, 7, 8, 9]

Pila horretan oinarrituz (baina pilaren ordez zerrendez osatutako zerrenda erabiliz) ordenatze azkarra inplementatzen duen funtzioa honako hau izango litzateke:

```
qs_lag :: [[Integer]] -> [Integer] -> [Integer]

qs_lag [ ] ord = ord
qs_lag (x:s) ord
        | (length x == 0) || (length x == 1)          = qs_lag s (ord ++ x)
        | otherwise       = qs_lag ([q1, [head x], q2] ++ s) ord
                                  where   q1 = [y | y <- (tail x), y <= (head x)]
                                          q2 = [y | y <- (tail x), y > (head x)]
```

Ordenatze azkarra bukaerako errekurtsibitatearekin burutzen duen funtzioa honako hau izango litzateke:

```
qs_be :: [Integer] -> [Integer]

qs_be r = qs_lag [r] []
```

Funtzio horrek, ordenatu beharreko r zerrenda pilan (zerrenden zerrendan) ipintzen du eta eraiki beharreko zerrenda ordenatua zerrenda hutsarekin hasieratzen du.

7.12.3. Nahasketa bidezko ordenazioa (merge sort): murgilketa eraginkortasuna lortzeko (bukaerako errekurtsibitatea) eta zerrenda-eraketa

Adibide honetan definitzen diren funtzioetan murgilketaren teknika eta zerrenda-eraketaren teknika aurki ditzakegu.

Nahasketaren bidezko ordenatze-metodoa honako era honetara azal daiteke:

- Zerrendaren luzera bikoitia baldin bada, luzera bereko bi zatitan banatu zerrenda. Zerrendaren luzera bakoitia baldin bada, zati batean bestean baino elementu bat gehiago ipiniz, bitan banatu zerrenda.
- Nahasketaren metodoa erabiliz bi azpizerrenda horiek ordenatu, bakoitza bere aldetik.
- Bi zerrenda horiek ordenatu ondoren, lortutako bi zerrenda ordenatu horiek nahastuz, ordenatutako zerrenda bakarra eraiki. Ordenatuta dauden bi zerrenda nahasteko, bi zerrendetako lehenengo elementuetatik txikiena hartu beharko da urrats bakoitzean, prozesua ordenatutako zerrenda bakarra lortu arte errepikatuz. Ordenatze-metodo hau ere "zatitu eta garaile izango zara" teknikaren beste kasu bat da.

Orain, [5, 8, 3, 9, 7, 2] zerrenda ordenatzeko beharko litzatekeen prozesua azalduko da:

1. Hasteko, [5, 8, 3, 9, 7, 2] zerrenda bi zatitan banatuko da: [5, 8, 3] y [9, 7, 2].

	Hasierako zerrenda [5, 8, 3, 9, 7, 2]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[5, 8, 3]		[9, 7, 2]

2. Orain [5, 8, 3] ordenatu behar da. Hori egiteko, zerrenda bitan banatuko da, [5] eta [8, 3] zerrendak lortuz. [5] zerrendak elementu bakarra duenez, ordenatuta dago. Beraz, [8, 3] zerrenda ordenatzea falta da.

	Hasierako zerrenda [5, 8, 3]	
Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[5] Elementu bakarra duenez, ordenatuta dago		[8, 3]

3. [8, 3] zerrenda ordenatzeko, bitan zatituko dugu, [8] eta [3] zerrendak lortuz. Bi zerrenda horiek elementu bakarrekoak direnez, ordenatuta daude. Ordenatutako bi zerrenda ditugun aldi bakoitzean, nahastu egin behar dira ordenatutako zerrenda bakarra eraikiz. Kasu honetan, [8] eta [3] zerrendak nahastuz [3, 8] zerrenda lortuko da.

Hasierako zerrenda
[8, 3]

Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[8] Elementu bakarra duenez, ordenatuta dago		[3] Elementu bakarra duenez, ordenatuta dago

Nahastu
[3, 8]

Beraz [8, 3] ordenatu da eta [3, 8] geratu da. Horretarako [8] eta [3] zerrendak nahastu dira. Zerrenda horiek nahasteko, 8 eta 3 balioetatik txikiena hartu da eta zerrenda berri bat eraikitzen hasi gara, [3] zerrenda berria geratu zaigularik. Jarraian [8] eta [] zerrendak nahastu dira. Bigarrena hutsa denez, emaitza [8] da eta orain arte eraikitako zerrenda [3] ++ [8] = [3, 8] da.

4. Orain 2. urratsean lortutako zerrenda biak ordenatuz lortutako [5] eta [3, 8] zerrendak ditugunez, nahastu egin behar dira. Bi zerrenda horiek nahasteko 5 eta 3ren artetik txikiena aukeratu behar da eta zerrenda berria eraikitzen hasi beharko da. Kasu honetan, zerrenda berri gisa momentuz [3] edukiko dugu. Hurrengo urrats bezala, [5] eta [8] zerrendak nahastu behar dira. Bi zerrenda horiek nahasteko 5 eta 8ren artetik txikiena aukeratu behar da eta aurreko urratsean eraiki dugun [3] zerrendari eskuinetik erantsi beharko diogu. Beraz, [3, 5] zerrenda daukagu orain. Bukatzeko, [] eta [8] zerrendak nahastu behar dira. Lehenengoa hutsa denez, [8] zerrenda lortuko da emaitza moduan eta guztira [3, 5] ++ [8] = [3, 5, 8] zerrenda geratuko zaigu.

5. [9, 7, 2] zerrenda ordenatzeko unea da orain. Zerrenda hori, 1. urratsean lortu den bigarren zerrenda da. Hasteko, bitan zatituko dugu, [9] eta [7, 2] zerrendak lortuz. [9] zerrenda ordenatuta dago. [7, 2] zerrenda ordenatzea falta zaigu.

Hasierako zerrenda
[9, 7, 2]

Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[9] Elementu bakarra duenez, ordenatuta dago		[7, 2]

6. [7, 2] zerrenda ordenatzeko, bi zerrendetan banatuko dugu, [7] eta [2] zerrendak lortuz. Bi zerrenda horiek elementu bakarrekoak direnez, ordenatuta daude.

Hasierako zerrenda
[7, 2]

Lehenengo zatia	Luzera bera edo ia luzera bera duten bi zatitan banatu	Bigarren zatia
[7] Elementu bakarra duenez, ordenatuta dago		[2] Elementu bakarra duenez, ordenatuta dago

Nahastu
[2, 7]

Beraz [7, 2] ordenatu dugu eta [2, 7] geratu zaigu. Horretarako [7] eta [2] zerrendak nahastu behar izan ditugu. Bi zerrenda horiek nahasteko, 7 eta 2 balioetatik txikiena hartu dugu eta zerrenda berri bat eraikitzen hasi gara, [2] zerrenda berria geratu zaigularik. Gero, [7] eta [] zerrendak nahastu dira. Bigarrena hutsa denez, emaitza [7] da eta orain arte eraikitako zerrenda [2] ++ [7] = [2, 7] da.

7. Beraz, 5. urratsean lortu diren zerrendak ordenatu dira eta [9] eta [2, 7] zerrendak lortu dira. Orain bi zerrenda horiek nahastu egin behar ditugu. Bi zerrenda horiek nahasteko 9 eta 2 zenbakietatik txikiena aukeratu behar dugu eta zerrenda berria eraikitzen hasi beharko dugu. Kasu honetan zerrenda berri gisa [2] edukiko dugu lehenengo urrats honen ondoren. Hurrengo urratsa, [9] eta [7] zerrendak nahastea izango da. Bi zerrenda horiek nahasteko 9 eta 7ren artetik txikiena aukeratu behar da eta aurreko urratsean eraiki dugun [2] zerrendari eskuinetik erantsi beharko diogu. Beraz, [2, 7] zerrenda daukagu orain. Bukatzeko, [9] eta [] zerrendak nahastu behar dira. Bigarrena hutsa denez, [9] zerrenda lortuko da emaitza gisa eta guztira [2, 7] ++ [9] = [2, 7, 9] zerrenda geratuko zaigu.

8. Une honetan, 1. urratsean lortu diren bi zerrendei dagozkien zerrenda ordenatuak ditugu: [3, 5, 8] eta [2, 7, 9]. Bi zerrenda horiek nahastu egin behar dira. Bi zerrenda horiek nahasteko 3 eta 2 balioak hartu eta txikiena aukeratu behar da zerrenda berri bat eraikitzen hasteko. Lehenengo urratsaren ondoren [2] zerrenda geratuko zaigu. Jarraian [3, 5, 8] eta [7, 9] zerrendak nahastuz joan behar dugu. Bi zerrenda horiek nahasteko, 3 eta 7 balioetatik txikiena hartu eta eraikitzen ari garen zerrenda berriari eskuinetik erantsi behar zaio. Beraz [2, 3] lortuko dugu. Hurrengo urratsa, [5, 8] eta [7, 9] zerrendak nahastea izango da. Horretarako, 5 eta 7 zenbakietatik txikiena aukeratu eta orain arte eraikita daukagun [2, 3] zerrendari erantsi beharko diogu eskuinetik. Beraz, [2, 3, 5] zerrenda izango dugu orain. Nahasketarekin aurrera jarraituz, [8] eta [7, 9] zerrendak ditugu nahasteko. Urrats honetan [2, 3, 5] zerrendari 8 eta 7ren artetik txikiena erantsi beharko diogu eta, ondorioz, [2, 3, 5, 7] lortuko dugu. Hurrengo urratsean [8] eta [9] nahastea da helburua. Beraz, 8 eta 9 zenbakiak hartuz, txikiena [2, 3, 5, 7] zerrendari erantsiko diogu, [2, 3, 5, 7, 8] zerrenda lortuz. Azkeneko urratsa, [] eta [9] nahastea izango da. Lehenengo zerrenda hutsa denez, nahasketaren emaitza [9] da eta guztira eraikitako zerrenda [2, 3, 5, 7, 8] ++ [9] = [2, 3, 5, 7, 8, 9] da.

Haskell erabiliz honela garatu dezakegu metodoa:

- Dagoeneko ordenatuta dauden bi zerrenda nahasten dituen *nahastu* izeneko funtzioa definitu.
- Honako ekintza hauek egiteaz arduratuko den funtzioa definitu:
    - Ordenatu beharreko zerrenda bi zatitan banatu (zati bakoitzak luzera bera izanda edo gehienez zati batek besteak baino elementu bat gehiago izanda).
    - Era errekurtsiboan ordenatu zati biak.

- Ordenatutako bi zatiak nahastu, nahasteaz arduratzen den funtzioari deituz.

Hasteko, ordenatuta dauden bi zerrenda emanda, zerrenda bietako elementuak nahastuz eraikitzen den zerrenda ordenatua itzuliko duen funtzioa definituko dugu:

```
nahastu :: [Integer] -> [Integer] -> [Integer]

nahastu [ ] r = r
nahastu (x:s) r
        | r == [ ]          = (x:s)
        | x <= (head r)     = x : (nahastu s r)
        | otherwise         = (head r) : (nahastu (x:s) (tail r))
```

Dagoeneko ordenatuta dauden bi zerrenda nahastuz ordenatuta dagoen zerrenda berria eraikitzen duen funtzio horrek ez du bukaerako errekurtsibitaterik. Bukaerako errekurtsibitatea izatea nahi izanez gero, emaitza gisa lortuz joango den zerrenda gordetzeko balio duen parametro berri bat ipini beharko genuke. Hasteko, funtzio laguntzailea definituko dugu eta gero bukaerako errekurtsibitatea duen eta bi zerrenda ordena errespetatuz nahasteko balio duen funtzioa definituko da:

```
nahastu_lag:: [Integer] -> [Integer] -> [Integer] -> [Integer]

nahastu_lag [] r q = q ++ r
nahastu_lag (x:s) r q
        | r == []           = q ++ (x:s)
        | x <= (head r)     = nahastu_lag s r (q ++ [x])
        | otherwise         = nahastu_lag (x:s) (tail r) (q ++ [head r])
```

```
nahastu_be:: [Integer] -> [Integer] -> [Integer]

nahastu_be r w = nahastu_lag r w []
```

Dagoeneko ordenatuta dauden bi zerrenda nahasteko balio duten bi funtzio definitu ditugu: *nahastu* eta *nahastu_be*. Bi funtzio horiek gauza bera egiten dute baina nahastu_be erabiliko dugu bukaerako errekurtsibitatea duelako. Orain, nahasketa bidezko ordenazioari, hau da, merge sort metodoari jarraituz zerrenda bat ordenatzen duen funtzioa defini dezakegu:

```
ms :: [Integer] -> [Integer]

ms [] = []
ms (x:s)
        | s == [] = [x]
        | otherwise         = nahastu_be (ms q1) (ms q2)
                                where   q1 = genericTake ((length (x:s)) `div` 2) (x:s)
                                        q2 = genericDrop ((length (x:s)) `div` 2) (x:s)
```

Dei errekurtsiboen bidez, ms funtzioak adar ugari dituen zuhaitz bitar bat eraikiko du. Prozesua xehetasun handiagoarekin aztertuz gero, azkenean elementu bakarreko zerrendak lortzen direla ikus dezakegu. Elementu bakarreko zerrendetara iritsitakoan

zerrenda horiek binaka nahasten hasiko da, hasierako zerrendari dagokion zerrenda ordenatua lortu arte. Hori kontuan hartuz, adarkatzea saihesten duen eta gainera bukaerako errekurtsibitatea duen funtzioa defini dezakegu Haskell lengoaiak zerrendekin lana egiteko eskaintzen dizkigun erraztasunak erabiliz. Horretarako, honako hauek egin beharko genituzke:

- Dagoeneko ordenatuta dauden bi zerrenda nahasten dituen funtzioa definitu.
- Dagoeneko ordenatuta dauden zerrendez osatutako zerrenda bat emanda, zerrenda horiek nahastuz joango den ms_lag funtzio laguntzailea definitu. Hasteko, lehenengo eta bigarren zerrendak nahastuko ditu. Nahasketa horretan lortutako zerrenda berria hirugarrenarekin nahastuko du eta abar. Zerrenda bakar batez osatutako zerrenda geratzen denean amaituko da prozesua.
- Ordenatu beharreko zerrenda bat emanda, zerrenda horretako elementuak osagai bakarreko zerrendatan ipiniz eta zerrenda horiek denak zerrenda batean sartuz lortzen den zerrenda ms_lag funtzioari pasatzeaz arduratuko den ms_be funtzioa definitu.

```
ms_lag :: [[Integer]] -> [Integer]

ms_lag [] = []
ms_lag (x:s)
        | s == [] = x
        | otherwise       = ms_lag (q : (tail s))
                                    where   q = nahastu_be x (head s)
```

Nahasketa bidezko ordenatzea bukaerako errekurtsibitatearekin inplementatzen duen funtzioa honako hau izango litzateke:

```
ms_be :: [Integer] -> [Integer]

ms_be r = ms_lag [[y] | y <- r]
```

7.12.4. Herbeheretar banderaren problema: zerrenda-eraketa erabiliz

(The problem of the Dutch National Flag, "A Discipline of Programming.", Edsger W. Dijkstra, 1975)

Herbeheretar bandera:

Agindu bidezko lengoaietan problema hau honela planteatzen da:

Har dezagun n osagaiko A(1..n) bektore bat eta demagun bektore horretako elementuak hiru multzotan sailka daitezkeela: gorriak, zuriak eta urdinak. Helburua,

bektorearen ezkerraldean gorriak diren elementu denak, erdian zuriak eta eskuinaldean urdinak lagatzea da.

Zenbaki osozko bektore bat kontsideratzen badugu, esate baterako gorria izatea zenbaki lehena izatearekin identifika dezakegu, zuria izatea bikoiti ez lehena izatearekin eta urdina izatea bakoiti ez lehena izatearekin.

Beste aukera batzuk ere badaude, adibidez, gorria izatea positiboa ez izatearekin identifika dezakegu (≤ 0), zuria izatea positiboa eta lehena izatearekin eta urdina izatea positiboa bai baina lehena ez izatearekin. Aurreko paragrafoan esandakoa jarraituko dugu guk hemen.

Ada edo Java erako lengoaietan problema honela ebatziko genuke:

```
sgleh := 1; /* Sailkatu gabeko lehenengoaren posizioa */
lehzur := 0; /* Lehenengo zuriaren posizioa */
lehurd := n + 1; /* Lehenengo urdinaren posizioa */

while (sgleh < lehurd) loop
        if gorria(A(sgleh))     /* lehena (A(sgleh)) */
        then    lehzur := lehzur + 1;
                lag := A(sgleh);
                A(sgleh) := A(lehzur);
                A(lehzur) := lag;
                sgleh := sgleh + 1;

        elsif zuria(A(sgleh))   /* bikoitia(A(sgleh)) */
        then    sgleh:= sgleh + 1;

        else    /* urdina(A(sgleh)), hau da, bakoitia(A(sgleh)) */
                lehurd:= lehurd - 1;
                lag := A(sgleh);
                A(sgleh) := A(lehurd);
                A(lehurd) := lag;
        end if;
end loop;
```

Algoritmo horrek orokorrean kolore bereko elementuen ordena alda dezake. Adibidez, (6, 8, 5, 15, 11, 3, 10) bektorea hartzen badugu, algoritmoak (5, 11, 3, 10, 8, 6, 15) bektorea itzuliko luke. Beraz, 6, 8 eta 10en arteko ordena aldatu egin da.

Algoritmo horretan, while-eko edozein bueltatan bektorearen egoera honako hau da:

Gorriak	Zuriak	Sailkatu gabeak	Urdinak
	↑ lehzur	↑ sgleh	↑ lehurd

Kolore bereko elementuen ordena mantentzeko, beste bi while txertatu beharko genituzke algoritmo horretan:

```
sgleh := 1; /* Sailkatu gabeko lehenengoaren posizioa */
lehzur := 0; /* Lehenengo zuriaren posizioa */
lehurd := n + 1; /* Lehenengo urdinaren posizioa */

while (sgleh < lehurd) loop
        if gorria(A(sgleh))     /* lehena (A(sgleh)) */
        then    lehzur := lehzur + 1;
                lag := A(sgleh);
                i := sgleh;
                while i >= lehzur loop
                        A(i) := A(i - 1);
                        i := i - 1;
                end loop;
                A(lehzur - 1) := lag;
                sgleh := sgleh + 1;

        elsif zuria(A(sgleh))   /* bikoitia(A(sgleh)) */
        then    sgleh:= sgleh + 1;

        else    /* urdina(A(sgleh)), hau da, bakoitia(A(sgleh)) */
                lehurd:= lehurd - 1;
                lag := A(sgleh);
                i := sgleh
                while i <= lehurd - 1 loop
                        A(i) := A(i + 1);
                        i := i + 1;
                end loop;
                A(n) := lag;
        end if;
end loop;
```

Algoritmo berri honek (6, 8, 5, 15, 11, 3, 10) bektorearentzat (5, 11, 3, 6, 8, 10, 15) itzuliko luke, kolore edo talde bereko elementuen arteko ordena errespetatuz.

Zerrenda-eraketaren teknika erabiliz, sailkapen hori burutzen duen funtzioa definitzea errazagoa da Haskell lengoaian. Funtzioari *hbp* deituko diogu. Funtzio horrek, kolore edo talde bereko elementuen arteko ordena mantenduko du:

```
hbp :: [Integer] -> [Integer]

hbp [] = []
hbp (x:s) = [y | y <- (x:s), lehena_ze y] ++ [y | y <- (x:s), not (lehena_ze y), y `mod` 2 == 0] ++
                [y | y <- (x:s), not (lehena_ze y), y `mod` 2 /= 0]
```

hbp [6, 8, 5, 15, 11, 3, 10] → [5, 11, 3, 6, 8, 10, 15]

Definizioan ikusten dea hbp funtzioa ez dela errekurtsiboa eta, horregatik, definizioa hobetu daiteke zerrenda hutsaren eta hutsa ez den zerrendaren kasuak bereizi gabe.

Gainera espresio luzeak daudenean, hobe da where erabiltzea, jarraian azalduko den moduan:

```
hbp2 :: [Integer] -> [Integer]

hbp2 r = gorriak ++ zuriak ++ urdinak
                 where   gorriak = [y | y <- r, lehena_ze y]
                         zuriak = [y | y <- r, not (lehena_ze y), y `mod` 2 == 0]
                         urdinak = [y | y <- r, not (lehena_ze y), y `mod` 2 /= 0]
```

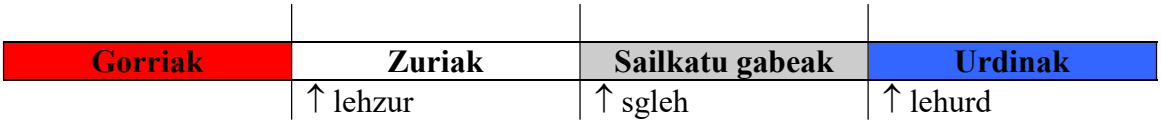
Hor r zerrenda hutsa baldin bada, *gorriak*, *zuriak* eta *urdinak* izeneko hiru zerrendak ere hutsak izango dira eta bukaerako emaitza ere zerrenda hutsa izango da.

7.12.5. Herbeheretar banderaren problema: murgilketa erabiliz

Atal honetan, herbeheretar banderaren problema ebazteko beste era bat azalduko da.

Aurreko atalean bezala, har dezagun n osagaiko A(1..n) bektore bat (Haskell-en bektorea eduki beharrean zerrenda bat edukiko dugu) eta demagun bektore horretako elementuak hiru multzotan sailka daitezkeela: gorriak, zuriak eta urdinak. Helburua, bektorearen ezkerraldean gorriak diren elementu denak, erdian zuriak eta eskuinaldean urdinak lagatzea da. Zenbaki osozko bektore bat kontsideratuko dugu eta gorria izatea zenbaki lehena izatearekin identifikatuko dugu, zuria izatea bikoiti ez lehena izatearekin eta urdina izatea bakoiti ez lehena izatearekin.

Intuitiboki, tarteko urrats batean gaudenean egoera honako hau izango da:



Murgilketaren teknika erabiliz, sailkapen hori burutzen duen funtzioa definituko da Haskell lengoaia erabiliz. Funtzioari *hbp_mg* deituko diogu. Funtzio horrek kolore edo talde bereko elementuen arteko ordena mantenduko du. Horretarako, une bakoitzean lau zerrenda edukiko dira: oraindik sailkatu gabe dauden elementuez osatutako zerrenda, dagoeneko sailkatuak izan diren elementu gorriez osatutako zerrenda, dagoeneko sailkatuak izan diren elementu zuriez osatutako zerrenda eta dagoeneko sailkatuak izan diren elementu urdinez osatutako zerrenda. Beraz, zerrenda bat edukitzetik lau zerrenda edukitzera pasatuko gara, hau da, parametro bat izatetik lau parametro izatera pasatuko gara. Lau parametro izango dituen funtzio laguntzaileari hbp_lag deituko diogu:

```
hbp_lag :: [Integer] -> [Integer] -> [Integer] -> [Integer] -> [Integer]

hbp_lag [] g z u= g ++ z ++ u
hbp_lag (x:s) g z u
        | lehena_ze x                                   = hbp_lag s (g ++ [x]) z u
        | (not (lehena_ze x)) && (x `mod` 2 == 0)       = hbp_lag s g (z ++ [x]) u
        | (not (lehena_ze x)) && (x `mod` 2 /= 0)       = hbp_lag s g z (u ++ [x])
```

Lehenengo parametroa oraindik sailkatu gabe dauden elementuez osatutako zerrenda da eta beste hiru parametroak, hurrenez hurren, dagoeneko sailkatuak izan diren elementu gorriez, zuriez eta urdinez osatutako zerrendak dira.

Bukatzeko, hbp_mg funtzioa hbp_lag funtzioari parametro egokiekin deitzeaz arduratuko da:

```
hbp_mg :: [Integer] -> [Integer]

hbp_mg q = hbp_lag q [] [] []
```

Hor, hasieran, dagoeneko sailkatuak izan diren elementu gorriez, zuriez eta urdinez osatutako zerrendak hutsak izango dira, oraindik ez delako elementurik sailkatu. Hasieran elementu denak lehenengo parametroan daude, hau da, q zerrendan.

7.13. Datu-mota berrien definizioa

Orain arte oinarrizko datu-motekin (Bool, Int, Integer, Char, Float, Double) eta datu-mota konposatuekin (zerrendak eta tuplak) aritu gara. Orain datu-mota berriak definitzeko hiru era azalduko dira.

7.13.1. Datu-mota sinonimoen definizioa type erabiliz

Batzuetan oinarrizko datu-moten, zerrenden eta tuplen bidez adieraz daitekeen datu-motaren bat erabili nahiko dugu baina behar bada ondo etorriko zaigu datu-mota horri izen desberdin bat ematea, izen horren datu-motaren esanahia hobeto adierazten duelako edo. Horrelakoetan, **type** erabiltzea da aukera onena. Izan ere, type mekanismoak aurretik existitzen diren datu-moten sinonimoak sortzea ahalbidetzen baitu.

Adibidez, hilabete bat adierazten duen zenbaki bat emanda, hurrengo hilabeteari dagokion zenbakia itzuliko duen funtzio bat nahi badugu, funtzio hori honela defini dezakegu:

```
hh :: Integer -> Integer

hh x
 | x < 0 || x > 12        = error "Datua ez da egokia."
 | x < 12                 = x + 1
 | x == 12                = 1
```

Funtzio baten motak funtzioaren esanahia eta egin beharrekoa hobeto ulertzeko balio behar du. hh funtzioaren kasuan, zenbaki oso bat emanda zenbaki oso bat itzuliko duela ikusteak ez du asko laguntzen funtzioaren esanahia ulertzen. Guk badakigu hilabete bat adierazten duen zenbaki oso bat eman nahi diogula eta funtzioak hurrengo hilabetearen zenbakia itzuliko duela. Informazio hori funtzioan bertan ipintzeko, Hilabete_zenb datu-mota defini dezakegu Integer datu-motaren sinonimo gisa type mekanismoa erabiliz:

```
type Hilabete_zenb = Integer

hh2 :: Hilabete_zenb -> Hilabete_zenb

hh2 x
 | x < 0 || x > 12        = error "Datua ez da egokia."
 | x < 12                 = x + 1
 | x == 12                = 1
```

Hilabete_zenb mota Integer motaren sinonimoa da. Datu-mota bera dira eta bata edo bestea erabil daiteke edozein definiziotan. Hilabete_zenb datu-mota definituz lortzen dugun gauza bakarra, hh2 funtzioaren esanahia ulertzen errazagoa izatea da: hilabete-zenbaki bat emango diogu eta hilabete zenbaki bat itzuliko du.

Honako funtzio honek, zenbaki osozko bi bikote emanda, bi bikote horien batura kalkulatuko du:

```
bikote_batura :: (Integer, Integer) -> (Integer, Integer) -> (Integer, Integer)

bikote_batura (x, y) (z, w) = (x + z, y + w)
```

Zenbaki osozko bikoteei izen bat emateko aukera eskaintzen digu type mekanismoak:

```
type Oso_bikote = (Integer, Integer)

bikote_batura2 :: Oso_bikote -> Oso_bikote -> Oso_bikote

bikote_batura2 (x, y) (z, w) = (x + z, y + w)
```

Orain definituko dugun *boskote_pos* izeneko funtzioak, 1 eta 5en arteko zenbaki oso bat eta t motako elementuz eratutako boskote bat emanda, zenbaki osoaren bidez adierazitako posizioko elementua itzuliko du.

```
boskote_pos :: Integer -> (t,t,t,t,t) -> t

boskote_pos p (v, w, x, y, z)
 | p < 0 || p > 5          = error "Posizioa ez da egokia."
 | p == 1                  = v
 | p == 2                  = w
 | p == 3                  = x
 | p == 4                  = y
 | p == 5                  = z
```

Funtzio horren motak funtzioaren esanahia hobeto adierazteko, Integer motaren sinonimoa den *Posizioa* mota (t, t, t, t, t) motaren sinonimoa den *Boskotea t* mota defini ditzakegu type erabiliz. Hori da, hain zuzen ere, boskote_pos2 funtzioaren definizioan egingo duguna:

```
type Posizioa = Integer

type Boskotea t = (t,t,t,t,t)

boskote_pos2 :: Posizioa -> Boskotea t -> t

boskote_pos2 p (v, w, x, y, z)
 | p < 0 || p > 5          = error "Posizioa ez da egokia."
 | p == 1                  = v
 | p == 2                  = w
 | p == 3                  = x
```

```
 | p == 4                = y
 | p == 5                = z
```

boskote_pos2 funtzioaren motak datuei buruzko informazio gehiago emateaz gain, bigarren argumentuaren kasuan t bost aldiz idaztea saihesten da. Izan ere, Boskotea t mota definitu ondoren nahikoa da t behin bakarraik idaztea.

Orain definituko dugun funtzioak, t1 motako elementu bat eta t1 motako elementu batez eta t2 motako lau elementuz osatutako boskotez eratutako zerrenda bat emanda, lehenengo osagaitzat lehenengo argumentu gisa emandako balioa duten boskotez eratutako zerrenda itzuliko du:

```
type Boskotea2 t1 t2 = (t1,t2,t2,t2,t2)

lehenengoa_berdina :: Eq t1 => t1 -> [Boskotea2 t1 t2] -> [Boskotea2 t1 t2]

lehenengoa_berdina e [] = []
lehenengoa_berdina e ((v, w, x, y, z):s)
 | e == v              = (v, w, x, y, z):(lehenengoa_berdina e s)
 | otherwise           = lehenengoa_berdina e s
```

Boskotea2 motak bi parametro ditu (t1 eta t2); izan ere, kontsideratu beharreko boskoteetan lehenengo osagaiaren mota eta beste lau osagaien mota desberdinak izan daitezke. Boskotea2 t1 t2 mota funtzioaren bigarren argumentua eta emaitza era horretako boskotez osatutako zerrendak direla adierazteko erabili da. Gainera, funtzioaren definizioan t1 motako bi elementu berdintzaren bidez konparatzen direnez, Eq t1 => baldintza ipini behar da.

Como último ejemplo, se muestra la definición de la función *tarjeta_credito_usuario* que dados un número de dni, una letra de dni y una lista de usuarios, devuelve el número de la tarjeta de crédito del usuario. Si no hay ningún usuario con el número de dni y la letra de dni dados, se muestra un mensaje de error:

```
type Nan_zenb = Integer
type Nan_letra = Char
type Txartel_zenb = Integer
type Pertsona = (Nan_zenb, Nan_letra, Txartel_zenb)
type Bezeroak = [Pertsona]

bezeroaren_kreditu_txartela :: Nan_zenb -> Nan_letra -> Bezeroak -> Txartel_zenb

bezeroaren_kreditu_txartela nan_z nan_l [] = error "Erregistratu gabeko bezeroa."

bezeroaren_kreditu_txartela nan_z nan_l ((nz, nl, tz):s)
 | nan_z == nz && nan_l == nl      = tz
 | otherwise                       = bezeroaren_kreditu_txartela nan_z nan_l s
```

Adibide horretan garbi ikusten da sinonimoak erabiltzeak asko errazten duela funtzioaren definizioan parte hartzen duten datuen esanahia ulertzen.

Laburbilduz, type mekanismoaren bidez definitutako motak type erabili gabe ere adieraz ditzakegun moten sinonimoak dira. Beraz, type ez da guztiz beharrezkoa baina datuen esanahia funtzioaren motan bertan ipintzeko balio du.

Oraintxe esan da type mekanismoak type erabili gabe ere adieraz daitezkeen motei izen bat emateko bakarrik balio duela. Ondorioz, type erabiliz ezin da mota errekurtsiborik definitu. Esate baterako, honako definizio errekurtsibo hau ez da onargarria:

```
type  Boskotea_errek t = (t, t, t, t, Boskotea_errek t))
```

Boskotea_errek t mota horren bidez, t motako lau osagai eta Boskotea_errek t motako bosgarren osagaia dituen boskoteak definitu nahi dira. Baina bosgarren boskote hori aldi berean t motako lau osagai eta Boskotea_errek t motako bosgarren osagaia dituen boskote bat izango da. Beraz, egitura infinitua definitzen ari gara. Baina Haskell-ek ez du onartuko type erabiliz mota hori definitzea.

7.13.2. Datu-mota berrien definizioa data erabiliz

Guztiz berria den datu-mota bat definitzeko **data** mekanismoa erabili behar da. Datu-mota finituak eta datu-mota infinituak defini daitezke data erabiliz. Datu-mota finituak definitzeko, mota horretako elementu denak izendatu beharko dira banan-banan. Datu-mota infinituak gehienetan errekurtsibitatearen bidez definituko dira.

Adibidez, urtaroen datu-mota defini dezakegu:

```
data Urtaroak =Negua | Udaberria | Uda | Udazkena
```

Urtaroak datu-mota, mota horretakoak diren balio denak banan-banan izendatuz definitu da. | marra bertikala balioak elkarrengandik bereizteko erabiltzen da. Balioen izenek, datu-moten izenek bezala, maiuskulaz hasi behar dute. Horrela definitutako balioak ez dira String motako karaktere-kateak. Urtaroak motako Negua balioa eta String motako "Negua" karaktere-katea gauza desberdinak dira. Balio boolear bat eta zenbaki oso bat konparaezinak diren era berean, Negua eta "Negua" ere ezin dira konparatu, ez baitira mota berekoak.

Datu-mota bat mota-horretakoak diren balio guztiak izendatuz definitzen dugunean, balio horientzat propietate edo ezaugarri batzuk ezarri ditzakegu:

- Mota horretako balioak berdintzaren bidez konparagarriak al diren ala ez adieraz dezakegu: **Eq**.
- Balio bat beste bat baino txikiagoa edo handiagoa al den galdetzerik ba al dagoen adieraz dezakegu, hau da, elementuen artean ordenarik ba al dagoen adieraz dezakegu: **Ord**.

- Datu-motako balioak karaktere-kate formatura itzuliak izan al daitezkeen (Negua baliotik "Negua" katea lortuz) eta karaktere-kate formatutik datu-motako baliora ("Negua" katetik Negua balioa lortuz) itzulpena egiterik ba al dagoen adieraz dezakegu. Horretarako, **Show** eta **Read** ezaugarriak ezarri beharko dira. Ezaugarri horiek garrantzitsuak dira bai balioak pantailan aurkeztu ahal izateko eta baita balioak pantailatik irakurri ahal izateko. Izan ere, pantailan agertzen den guztia Char edo String motakoa izango da.
- Datu-motako balioak zenbagarriak izatea nahi al dugun ere adieraz dezakegu. Datu-mota bateko balioak zenbagarriak badira, mota horretako bi balio emanda, bi balio horien artean dauden balioez eratutako zerrenda era automatikoan lortu ahal izango da. Horretako **Enum** ezaugarria ezarri beharko da. Adibidez, Urtaroak motak Enum ezaugarria baldin badu, [Negua .. Udazkena] espresioaren bidez Negua eta Udazkena balioen artean dauden balio guztiez osatutako zerrenda sortuko da automatikoki:

  [Negua, Udaberria, Uda, Udazkena]

  Era berean, [Udaberria .. Udazkena] espresioaren bidez Udaberria eta Udazkena balioen artean dauden balio guztiez osatutako zerrenda sortuko da automatikoki:

  [Udaberria, Uda, Udazkena]

- **Bounded** ezaugarriaren ezartzen bada, datu-motako balio txikiena eta balio handiena minBound eta MaxBound funtzioen bidez ezagutzeko aukera izango dugu. Esate baterako, Urtaroak datu-motak Bounded ezaugarria baldin badu, balio txikiena zein den jakiteko honako hau idatziko dugu:

  minBound :: Urtaroak

  Eta balio handiena zein den jakiteko honako hau idatziko dugu:

  maxBound :: Urtaroak

  Erantzunak Negua eta Udazkena izango lirateke hurrenez hurren.

Aipatu diren ezaugarri horiek datu-mota definitzean ezarri behar dira. Beraz, Urtaroak datu-motaren definizioa honako hau izango litzateke:

```
data Urtaroak = Negua | Udaberria | Uda | Udazkena
  deriving (Eq, Ord, Show, Read, Enum, Bounded)
```

Ezaugarri horiek ezartzeko **deriving** hitza ipini behar da eta, gero, parentesi artean, ezarri nahi diren ezaugarriak. Kasu honetan, aipatutako ezaugarri guztiak ezarri dira.

Datu-motako balioei emandako izenei (Negua, Udaberria, …) *funtzio edo eragiketa eraikitzaileak* deitzen zaie.

Urtaroen izenekin egin den era berean, datu-mota gehiago defini daitezke asteko egunen izenekin, hilabeteen izenekin, koloreen izenekin eta abar.

Eragiketa eraikitzaileek parametro gisa beste datu-motak edukitzea ere ahalbidetzen du **data** mekanismoak. Esate baterako, hiru irudi geometrikoz eratutako *Irudiak* mota defini dezakegu. Irudi horiek honako hauek izango dira:

- Zirkulua: bereizgarri gisa erradioa izango du.
- Karratua: bereizgarri gisa alde baten luzera izango du.
- Laukizuzena: bereizgarri gisa oinarria eta altuera izango ditu.

```
data Irudiak = Zirkulua Float | Karratua Float | Laukizuzena Float Float
  deriving (Eq, Ord, Show, Read)
```

Funtzio eraikitzaileak Zirkulua, Karratua eta Laukizuzena dira. Zirkulua eta Karratua funtzio eraikitzaileek parametro bakarra dute (Float) eta Laukizuzena funtzio eraikitzaileak bi parametro ditu (Float eta Float).

Irudiak izeneko datu-mota definitzean ez dira ezarri Enum eta Bounded ezaugarriak. Izan ere, ezaugarri horiek parametrorik ez duten funtzio eraikitzaileak bakarrik dituzten datu-mota finituentzat ezarri daitezke soilik.

Orain irudi baten azalera kalkulatuko duen funtzioa definituko dugu:

```
azalera :: Irudiak -> Float

azalera (Zirkulua erradioa) = pi * (erradioa ^ 2)
area (Karratua alde_luzera) = alde_luzera * alde_luzera
area (Laukizuzena oinarria altuera) = oinarria * altuera
```

Adibidez,

```
azalera (Karratua 3.5)
```

kasuan erantzuna `12.25` izango da eta

```
azalera (Laukizuzena 3.2 5)
```

kasuan erantzuna `16.0` izango da.

Datu-mota errekurtsiboak ere defini daitezke **data** mekanismoarekin. Hasteko, zenbaki arrunten edo naturalen mota definituko dugu:

```
data Nat = Zero | H Nat
  deriving (Eq, Ord, Show, Read)
```

Definizio horretan, Zero eta H dira funtzio eraikitzaileak. H horrek 'hurrengoa' adierazten du. Zero funtzio eraikitzaileak ez du argumenturik edo parametrorik baina H

funtzio eraikitzaileak Nat parametroa du. Alde batetik, Zero funtzio eraikitzaileak 0 balioaren baliokidea den balioa sortzen du. Bestalde, H funtzio eraikitzaileak unitate bat gehitzen dio emandako zenbaki arrunt bati, hau da, Nat motako elementu bati. Beraz, H Nat espresioak honako hau adierazten du: zeroren desberdinak diren zenbaki arruntak (edo naturalak) beste zenbaki arrunt bati H funtzio eraikitzailea erantsiz sortzen dira. Adibidez, bat zenbakia H Zero eran adieraziko genuke, bi zenbakia H (H Zero) eran, hiru zenbakia S (S (S Zero)) izango litzateke eta abar. Nat datu-mota errekurtsiboa da, zenbaki arrunt bakoitza (Nat motako elementu bakoitza) aurreko zenbaki arruntari H erantsiz definitzen delako. Eta hori H Nat espresioaren bidez formalizatzen da. Oro har, m balioa Nat motako elementu bat baldin bada, orduan H m hurrengo elementua da eta H (H m) hurrengoaren hurrengoa da eta abar. Nat datu-mota definitzean ez dira ezarri Enum eta Bounded ezaugarriak; izan ere, ezaugarri horiek parametrorik ez duten funtzio eraikitzaileak bakarrik dituzten datu-mota finituentzak dira soilik.

Nat datu-mota erabiltzeko, zenbaki arrunten gaineko eragiketak geuk definitu beharko ditugu. Esate baterako, bi zenbaki arrunten arteko batura honela definituko genuke:

```
batura_nat :: Nat -> Nat -> Nat

batura_nat Zero y = y
batura_nat (H x) y = H (batura_nat x y)
```

Adibidez,

```
batura_nat (H (H Cero)) (H (H (H Zero)))
```

kasuan erantzuna honako hau izango litzateke:

```
H (H (H (H (H Zero)))).
```

Datu-mota horrek datu-mota errekurtsiboak nola definitzen diren erakusteko balio digu baina Nat datu-mota bera ez da oso praktikoa; izan ere, zenbaki arruntak adierazteko era oso konplikatua da.

Nat datu-motak ez du parametrorik (nahiz eta H funtzio eraikitzaileak parametroa izan). Baina parametro bat edo gehiago dituzten datu-motak ere defini daitezke.

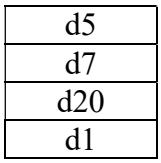
Orain *Pila t* datu-mota definituko dugu. Datu-mota hau polimorfikoa da, t parametroak edozein datu-mota adierazten duelako. Horrela, t parametroa datu-mota konkretu batez ordeztu dezakegu eta datu-mota konkretu horretako elementuak dituen pila bat lortuko dugu. Pila t datu-motaren definizioa honako hau da:

```
data Pila t = Phutsa | Pilaratu t (Pila t)
  deriving (Eq, Ord, Show, Read)
```

Funtzio edo eragiketa eraikitzaileak Phutsa eta Pilaratu dira. Phutsa funtzioak ez du argumenturik baina Pilaratu funtzioak bi argumentu ditu: lehenengoa t motakoa eta bigarrena Pila t motakoa. Phutsa funtzioak pila hutsa adierazten du edo sortzen du eta Pilaratu t (Pila t) espresioak t motako elementu bat t motako pila baten gainean ipiniz lortzen den pila adierazten du.

Grafikoki, pila hutsa honela irudikatu dezakegu:

Hainbat elementu dituen pila honela irudikatuko genuke:



Irudi horretan, d1, d20, d7 eta d5 t motako balio konkretuak dira.

Lau elementu dituen pila hori adierazteko era formala honako hau da:

Pilaratu d5 (Pilaratu d7 (Pilaratu d20 (Pilaratu d1 Phutsa)))

Espresio horren bidez honako hau adierazten da: pila hori eraikitzeko, hasteko pila hutsa sortuko litzateke, gero d1 balioa pilaratuko genuke, gero d20 balioa pilaratuko genuke, gero d7 pilaratuko genuke eta, bukatzeko, d5 pilaratuko genuke.

Orain pilen gaineko funtzio bat definituko dugu adibide gisa. Demagun zenbaki osozko pila bat dugula eta pila osatzen duten zenbaki osoen batura kalkulatu nahi dugula:

```
pila_batura :: Pila Integer -> Integer

pila_batura Phutsa = 0
pila_batura (Pilaratu x s) = x + (pila_batura s)
```

Kasu horretan, zenbaki osozko pilekin aritu nahi dugunez, t parametroa Integer motaz ordeztu dugu. Adibidez,

```
Pila_batura (Pilaratu 4 (Pilaratu 8 (Pilaratu 2 Phutsa)))
```

Kasuan erantzuna 14 izango da.

Baina edozein motentzat balio duten funtzioak ere defini ditzakegu. Hori lortzeko, t parametroa ez dugu ordeztuko, t bera mantenduko dugu.

Honako funtzio honek, t motako pila bat emanda, pilako elementu kopurua kalkulatuko du:

```
altuera :: Pila t -> Integer

altuera Phutsa = 0
altuera (Pilaratu x s) = 1 + (altuera s)
```

Pilako elementuen motak ez du eraginik altuera kalkulatzean. Adibidez,

```
altuera (Pilaratu 'a' (Pilaratu 'h' (Pilaratu 'w' Phutsa)))
```

kasuan, Char motako pila bateko elementu kopurua kalkulatuko da. Erantzuna 3 izango da.

Orain definituko dugun funtzioak, x zenbaki oso bat eta t motako pila bat emanda, pilaren gailurretik hasi eta lehenengo x elementuak ezabatuz lortzen den pila itzuliko du:

```
hainbeste_ezabatu :: Integer -> Pila t -> Pila t

hainbeste_ezabatu x Phutsa
 | x == 0        = Phutsa
 | otherwise     = error "Datua ez da egokia."

hainbeste_ezabatu x (Pilaratu y s)
 | x < 0         = error "Datua ez da egokia."
 | x == 0        = Pilaratu y s
 | otherwise     = hainbeste_ezabatu (x - 1) s
```

Adibidez,

```
ezabatu_hainbeste 2 (Pilaratu 5 (Pilaratu 4 (Pilaratu 8 (Pilaratu 9 Phutsa))))
```

kasuan erantzuna honako hau izango litzateke:
`Pilaratu 8 (Pilaratu 9 Phutsa)`.

Orain definituko den funtzioak, t motako x elementu bat eta t motako pila bat emanda, x elementuaren agerpen guztiak ezabatuko ditu pilatik:

```
elementua_ezabatu :: Eq t => t -> Pila t -> Pila t

elementua_ezabatu x Phutsa = Phutsa

elementua_ezabatu x (Pilaratu y s)
 | x == y        = elementua_ezabatu x s
 | otherwise     = Pilaratu y (elementua_ezabatu x s)
```

Definizio horretan Eq t => baldintza beharrezkoa da; izan ere, t motako elementuak berdintzaren bidez konparatzen dira.

Adibidez,

```
elementua_ezabatu 4 (Pilaratu 5 (Pilaratu 4 (Pilaratu 8 (Pilaratu 9 Phutsa))))
```

kasuan
`Pilaratu 5 (Pilaratu 8 (Pilaratu 9 Phutsa))`
erantzuna lortuko da.

Orain ilaren datu-mota definituko da. Ilara bat zerrenda baten antzekoa da, baina elementuak eskuineko ertzetik erantsi behar dira. Ilarako lehenengo elementua ezkerreko ertzekoa izango da eta azkeneko elementua eskuineko ertzekoa izango da.

```
data Ilara t = Ihutsa | Ipini (Ilara t) t
  deriving (Eq, Ord, Show, Read)
```

Orain, ilara bat emanda, hurrenez hurren, ilarako lehenengo eta azkeneko elementua itzuliko duten i_lehenengoa eta i_azkena funtzioak definituko ditugu:

```
i_lehenengoa :: Eq t => Ilara t -> t

i_lehenengoa Ihutsa = error "Ilara hutsa."

i_lehenengoa (Ipini s x)
 | s == Ihutsa  = x
 | otherwise    = i_lehenengoa s
```

```
i_azkena :: Ilara t -> t

i_azkena Ihutsa = error "Ilara hutsa."

i_azkena (Ipini s x) = x
```

Berdintzaren bidezko konparazio bat dagoenez i_lehenengoa funtzioaren definizioan, Eq t => baldintza behar da.

Adibidez, `i_lehenengoa (Ipini (Ipini (Ipini Ihutsa 9) 7) 5)` kasuan erantzuna 9 izango litzateke eta `i_azkena (Ipini (Ipini (Ipini Ihutsa 9) 7) 5)` kasuan erantzuna 5 izango litzateke.

Bi ilara emanda, i_elkartu funtzioak ilara biak elkartuz lortzen den ilara itzuliko du:

```
i_elkartu :: Ilara t -> Ilara t -> Ilara t

i_elkartu r Ihutsa = r

i_elkartu r (Ipini s x) = Ipini (i_elkartu r s) x
```

Adibidez,

```
i_elkartu (Ipini (Ipini Ihutsa 9) 7)  (Ipini (Ipini Ihutsa 0) 20)
```

kasuan erantzuna honako hau izango litzateke:

```
Ipini (Ipini (Ipini (Ipini Ihutsa 9) 7) 0) 20.
```

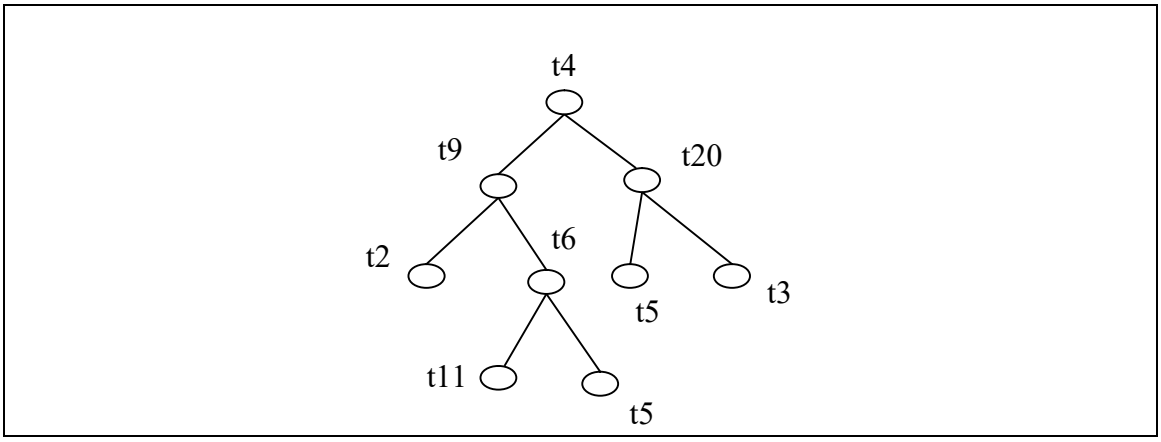
Datu-egiturekin erlazionatutako beste datu-mota bat zuhaitz bitarren datu-mota da. Datu-mota horren definizioa honako hau da:

```
data Zuhbit t = Zhutsa | Eraiki t (Zuhbit t) (Zuhbit t)
  deriving (Eq, Ord, Show, Read)
```

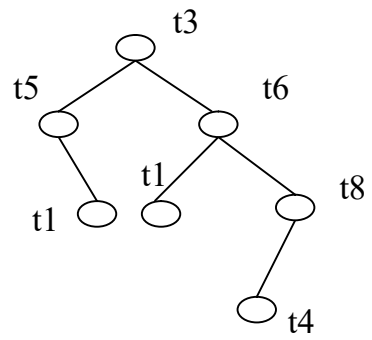
Definizio horren bidez honako t motako zuhaitz bitar bat bi eratakoa izan daitekeela adierazten da:

- hutsa, edo
- t motako elementu bat (erroa) eta t motako bi azpizuhaitzez (ezkerrekoa eta eskuinekoa) eratutakoa.

Zuhaitz bitar bat honela irudikatu daiteke:



Hor, t4 elementua erroa da, t9, t2, t6, t11 eta t5 elementuek ezkerreko azpizuhaitza osatzen dute eta t20, t5 eta t3 elementuek eskuineko azpizuhaitza osatzen dute. Beraz, zuhaitz bitarra t motako balioa duten adabegiz edo nodoz eta adabegien arteko arkuez osatuta dago. Adabegi bakoitzak aurreko bat (gurasoa) eta zero, bat edo bi ondorengo (une) ditu. Erroa da aurrekorik (gurasorik) ez duen bakarra. Beheko adabagiei, hau da, t2, t11, t5, t5 eta t3 adabegiei hostoak deitzen zaie. Eta grafikoki ez bada ikusten ere, hosto bakoitza ezkerreko azpizuhaitz hutsa eta eskuineko azpizuhaitz hutsa dituen azpizuhaitz baten erroa da. Hurrengo irudi horretan erakusten da zuhaitz bitarrak irregularragoak ere izan daitezkeela:



Zuhaitz bitar hori era formalean honela adieraziko genuke:

Eraiki t3 (Eraiki t5 Zhutsa (Eraiki t1 Zhutsa Zhutsa))
(Eraiki t6 (Eraiki t1 Zhutsa Zhutsa)
(Eraiki t8 (Eraiki t4 Zhutsa Zhutsa) Zhutsa))

Honako funtzio honek, zuhaitz bitar bat emanda, zuhaitz horretako elementu kopurua kalkulatuko du.

```
elem_kop :: Zuhbit t -> Integer

elem_kop Zhutsa = 0

elem_kop (Eraiki x a b) = 1 + (elem_kop a) + (elem_kop b)
```

Adibidez,

```
elem_kop (Eraiki 5 (Eraiki 20 Zhutsa (Eraiki 8 Zhutsa Zhutsa)) (Eraiki 10 Zhutsa Zhutsa))
```

kasuan erantzuna 4 izango da, zuhaitz bitar horretan lau elementu daudelako: 5, 20, 8 y 10.

La siguiente función, dados un elemento x de tipo t y un árbol binario de tipo t, elimina los subárboles de aquellos nodos cuyo valor es x. Eliminar un subárbol significa, realmente, sustituirlo por el árbol vacío:

```
azpizuhaitzak_ezabatu :: Eq t => t -> Zuhbit t -> Zuhbit t

azpizuhaitzak_ezabatu x Zhutsa = Zhutsa

azpizuhaitzak_ezabatu x (Eraiki y a b)
```

```
| x == y          = Eraiki y Zhutsa Zhutsa
| otherwise       = Eraiki y (azpizuhaitzak_ezabatu x a) (azpizuhaitzak_ezabatu x b)
```

Adibidez,

```
azpizuhaitzak_ezabatu 20 (Eraiki 5 (Eraiki 20 Zhutsa (Eraiki 8 Zhutsa
Zhutsa)) (Eraiki 10 Zhutsa Zhutsa))
```

kasuan erantzuna honako hau izango litzateke:

```
Eraiki 5 (Eraiki 20 Zhutsa Zhutsa) (Eraiki 10 Zhutsa Zhutsa)
```

Parametro bat baino gehiago dituzten datu-motak ere defini daitezke. Orain, aurretik definituta dauden bi datu-mota bilduz lortzen den datu-mota sortzeko bidea azalduko da:

```
data Bildura t1 t2 = A t1 | B t2
  deriving (Eq, Ord, Show, Read)
```

Definizio hori kontuan hartuz, esate baterako Integer eta Char motak bilduz sortzen den datu-mota erabil dezakegu. Bildura Integer Char datu-motaren kasuan, datu-mota horretako elementu batzuk honako hauek izango lirateke: A 20, A 405, A (-7), B 'd', B 'M', B ')' eta abar. Beraz, Bildura Integer Char datu-motako elementuak "A eta gero zenbaki oso bat" edo "B eta gero karaktere bat" erakoak dira.

Orain definituko den funtzioak, Bildura Integer Char motako elementu bat emanda, hurrengo elementua itzuliko du. Elementua Integer motako zenbaki bat baldin bada, hurrengo zenbakia itzuliko du eta elementua Char motako elementu bat baldin bada, hurrengo karakterea itzuliko du:

```
hurrengoa :: Bildura Integer Char -> Bildura Integer Char

hurrengoa (A x) = A (x + 1)

hurrengoa (B x) = B (succ x)
```

Definizio horretan succ, karaktere bat emanda, hurrengo karakterea itzultzen duen funtzioa da.

Adibidez

```
hurrengoa (A 48)
```

kasuan erantzuna honako hau izango da:

```
A 49
```

Eta

```
hurrengoa (B 'L')
```

kasuan erantzuna honako hau izango da:

```
B 'M'.
```

7.13.3. Parametro bat gehienez eta argumentu bakarreko funtzio eraikitzailea duten datu-mota berrien definizioa: newtype

Datu-mota berri bat definitzeko balio duen **newtype** mekanismoa, **data** mekanismoaren kasu partikular bat da. Datu-mota bat newtype erabiliz definitu ahal izateko, honako bi baldintza hauek bete behar dira:

- Datu-motak gehienez parametro bat edukitzea.
- Funtzio eraikitzaile bakarra edukitzea eta, gainera, funtzio eraikitzaile horrek zehazki argumentu bat edukitzea.

newtype erabiliz defini daitekeen edozein datu-mota **data** erabiliz ere defini daiteke, baina newtype erabiliz definitutako datu-motak eraginkortasun handiagoarekin maneiatzen ditu Haskell-en konpiladoreak. Izan ere, parametro bat gehienez eta zehazki argumentu bat duen funtzio eraikitzaile bakarra edukitzea betetzen duten datu-motak modu oso eraginkorrean kudeatzeko prestatuta dago Haskell-en konpiladorea. Beraz, definitu nahi dugun datu-motak ezaugarri horiek (parametro bat gehienez eta zehazki argumentu bat duen funtzio eraikitzaile bakarra) baldin baditu, hobe da newtype erabiltzea.

Orain zenbaki osozko multzoen mota definituko dugu:

```
newtype Multzoa = Mul [Integer]
 deriving (Eq, Ord, Show, Read)
```

Datu-mota horrek newtype erabili ahal izateko baldintzak betetzen ditu. Alde batetik, gehienez parametro bat eduki behar du eta Multzoa motak ez du parametrorik. Beste aldetik, eraikitzaile bakarra eduki behar du (Mul da eraikitzailea) eta eraikitzaile horrek zehazki argumentu bat eduki behar du; kasu honetan [Integer] da argumentu hori.

Multzoa datu-mota zerrendetan oinarrituta dago. Multzoa datu-motaren definizioa kontuan hartuz, multzo bat aurretik Mul hitza duen zenbaki osozko zerrenda bat da. Haskell lengoaian zerrendak erabiltzeko erraztasun asko dagoenez, multzoak zerrendatan oinarrituta egoteak onura eta erraztasun asko ekarriko dizkigu.

Adibidez, zenbaki oso bat eta zenbaki osozko multzo bat emanda, zenbakia multzo horretakoa al den erabakiko duen funtzioa honela defini dezakegu:

```
multzokoa_da :: Integer -> Multzoa -> Bool

multzokoa_da x (Mul s) = x `elem` s
```

Multzoek Mul s egitura dute eta, hor, s zenbaki osozko zerrenda bat da. Beraz, x zenbakia Mul s multzokoa al den jakiteko, nahikoa da x zenbakia s zerrendakoa al den erabakitzea.

Adibidez, `multzokoa_da 4 (Mul [5,7,8,7,20])` kasuan erantzuna `False` izango da.

Zenbakizko bi multzo biltzen dituen funtzioa honela defini dezakegu:

```
bildura :: Multzoa -> Multzoa -> Multzoa

bildura (Mul r) (Mul s) = Mul (r ++ s)
```

Definizio horretan, r eta s zerrendak direnez, zerrendak elkartzeko balio duen ++ eragiketa erabil dezakegu bi zerrenda horiek elkartzeko. Zerrendak elkartu ondoren lortu den zerrendari aurretik Mul funtzio eraikitzailea ipin badiogu Multzoa motako elementu bat lortuko dugu.

Adibidez,

```
bildura (Mul [5,8,3,9]) (Mul [2,2])
```

kasuan, erantzuna honako hau izango da:

```
Mul [5,8,3,9,2,2].
```

Zenbakizko bi multzoren arteko ebakidura kalkulatzen duen funtzioa honela defini dezakegu:

```
ebakidura :: Multzoa -> Multzoa -> Multzoa

ebakidura (Mul []) (Mul s) = Mul []
ebakidura (Mul (x:r)) (Mul s)
| s == []              = Mul []
| x `notElem` s        = ebakidura (Mul r) (Mul s)
| x `elem`s            = Mul (x: w)
                            where (Mul w) = (ebakidura (Mul r) (Mul s))
```

Definizio horretan, r eta s zerrendak direnez, zerrenden gaineko `elem` eta `notElem` funtzioak erabil daitezke. Gainera, Además, where klausula ere erabili da. Klausula hori erabiliz, Mul r eta Mul s multzoen arteko ebakidura gisa lortzen den multzoko zerrendari w izena ematen zaio.

Adibidez

```
ebakidura (Mul [5,8,3,9]) (Mul [2,2])
```

kasuan, erantzuna honako hau izango da:

```
Mul [].
```

Bestalde,

```
ebakidura (Mul [5,8,3,9]) (Mul [3,2,9,3,1])
```

kasuan, erantzuna honako hau izango da:

```
Mul [3,9].
```

Eta

```
ebakidura (Mul [5,8,3,9,3]) (Mul [2,9,3,1])
```

kasuan, erantzuna honako hau izango da:

```
Mul [3,9,3].
```

Matematikoki, errepikatutako elementuek eta elementuen ordenak ez du eraginik multzoetan. Beraz, Mul [5, 9, 20, 5] eta Mul [9, 9, 5, 20] espresioek multzo bera adierazten dute. Multzoa datu-mota definitzeko zerrendetan oinarritu garenez, errepikatuta dauden elementuak ez dira kontrolatzen. Bildura, ebakidura eta antzeko eragitekin lortzen diren emaitzetan errepikatutako elementurik ez dadin egon, errepikatutako elementuak ezabatzen dituen honako funtzio hau defini dezakegu:

```
forma_normala :: Multzoa -> Multzoa

forma_normala (Mul []) = Mul []
forma_normala (Mul (x:s))
 | x `elem` s            = forma_normala (Mul s)
 | x `notElem`s          = Mul (x: w)
                                where (Mul w) = (forma_normala (Mul s))
```

Funtzio horrek eskuineko ertzeko kopia mantentzen du (eskuinena dagoen kopia).

Adibidez,

```
forma_normala (Mul [8,9,5,9])
```

kasuan erantzuna honako hau da:

```
Mul [8,5,9].
```

Eta

```
forma_normala (Mul [8,9,5])
```

kasuan erantzuna honako hau da:

```
Mul [8,9,5].
```

Orain bildura eta ebakidura berriro definituko ditugu itzultzen duten erantzunean errepikatutako elementurik egon ez dadin. Beraz, erantzuna forma normalean egongo da.

```
bildura_fn :: Multzoa -> Multzoa -> Multzoa

bildura_fn x y = forma_normala (bildura x y)
```

Adibidez,

```
bildura_fn (Mul [5,8,3,9,3]) (Mul [2,9,3,1])
```

kasuan erantzuna honako hau izango da:

```
Mul [5,8,2,9,3,1]
```

Aldiz,

```
bildura (Mul [5,8,3,9,3]) (Mul [2,9,3,1])
```

kasuan erantzuna honako hau izango litzateke:

```
Mul [5,8,3,9,3,2,9,3,1]
```

Forma normala erabiltzen denean agertzen ez diren errepikatutako elementuak azpimarratu egin dira hor.

```
ebakidura_fn :: Multzoa -> Multzoa -> Multzoa

ebakidura_fn x y = forma_normala (ebakidura x y)
```

Adibidez,

```
ebakidura_fn (Mul [5,8,3,9,3]) (Mul [2,9,3,1])
```

kasuan erantzuna honako hau izango da:

```
Mul [9,3]
```

Aldiz,

```
ebakidura (Mul [5,8,3,9,3]) (Mul [2,9,3,1])
```

kasuan erantzuna beste hau izango litzateke:

```
Mul [3,9,3]
```

Forma normala erabiltzen denean agertzen ez den errepikatutako elementua azpimarratu egin da hor.

Multzoa izeneko mota zenbaki osoentzat bakarrik da. Edozein motatako elementuz eratutako multzoekin aritu nahi badugu, definizioa orokortu egin beharko da:

```
newtype Multzoa_orok t = Mulorok [t]
 deriving (Eq, Ord, Show, Read)
```

Multzoa_orok datu-motak parametro bat du (t) eta eragiketa eraikitzaile bakarra du (Mulorok). Eragiketa eraikitzaile horrek zehazki argumentu bat du ([t]).

Lehen Multzoa datu-motarentzat definitu ditugun lau eragiketak Multzoa_orok datu-motarentzat egokituko ditugu orain:

```
multzokoa_da_orok :: Eq t => t -> Multzoa_orok t -> Bool

multzokoa_da_orok x (Mulorok s) = x `elem` s
```

Multzo orokortuek Mulorok s egitura dute eta, hor, s t motako zerrenda bat da. Beraz, x elementua Mulorok s multzokoa al den jakiteko, nahikoa da x elementua s zerrendakoa al den erabakitzea. Aurredefinitutako `elem` funtzioa erabiltzeak berdintza erabiliz egindako konparaketak dakartza berarekin eta, horregatik, Eq t => baldintza ipini behar da.

Adibidez,

```
multzokoa_da_orok True (Mulorok [False, False, True, True])
```

kasuan erantzuna `True` izango da.

.

Eta

```
multzokoa_da_orok (5,8) (Mulorok [(9,0),(10,2),(5,8),(4,16),(0,0)])
```

kasuan erantzuna izango `True` da.

Edozein t motatako bi multzo biltzen dituen funtzioa honela defini dezakegu:

```
bildura_orok :: Multzoa_orok t -> Multzoa_orok t -> Multzoa_orok t

bildura_orok (Mulorok r) (Mulorok s) = Mulorok (r ++ s)
```

Edozein t motatako bi multzoren arteko ebakidura kalkulatzen duen funtzioa honela defini dezakegu:

```
ebakidura_orok :: Eq t => Multzoa_orok t -> Multzoa_orok t -> Multzoa_orok t

ebakidura_orok (Mulorok []) (Mulorok s) = Mulorok []
ebakidura_orok (Mulorok (x:r)) (Mulorok s)
 | s == []              = Mulorok []
 | x `notElem` s        = ebakidura_orok (Mulorok r) (Mulorok s)
 | x `elem`s            = Mulorok (x: w)
                        where (Mulorok w) = (ebakidura_orok (Mulorok r) (Mulorok s))
```

Definizio horretan (==) berdintza erabiltzen da eta gainera, berdintza behar duten aurredefinitutako `elem` eta `notElem` funtzioak ere erabiltzen dira. Beraz, Eq t => baldintza behar da.

Adibidez,

```
ebakidura_orok (Mulorok [True,False,True]) (Mulorok
[False,False,True])
```

kasuan erantzuna honako hau izango da:

```
Mulorok [True,False,True].
```

Eragiketa horiekin lortzen diren emaitzetan errepikatutako elementurik ez agertzeko, errepikapenak ezabatzen dituen funtzioa definitu beharko da Multzoa_orok datu-motarentzat.

```
forma_normala_orok :: Eq t => Multzoa_orok t -> Multzoa_orok t

forma_normala_orok (Mulorok []) = Mulorok []
forma_normala_orok (Mulorok (x:s))
 | x `elem` s              = forma_normala_orok (Mulorok s)
 | x `notElem`s           = Mulorok (x: w)
                            where (Mulorok w) = (forma_normala_orok (Mulorok s))
```

Adibidez,

```
forma_normala_orok (Mulorok ['b', 'v', 'a', 'a', 'b', 'e'])
```

kasuan erantzuna honako hau izango da:

```
Mulorok "vabe".
```

Orain edozein t motatako multzoentzat bilduraren eta ebakiduraren emaitza forma normalean (errepikapenik gabe) itzultzen duten funtzioak definituko ditugu.

```
bildura_orok_fn :: Eq t => Multzoa_orok t -> Multzoa_orok t -> Multzoa_orok t

bildura_orok_fn x y = forma_normala_orok (bildura_orok x y)
```

```
ebakidura_orok_fn :: Eq t => Multzoa_orok t -> Multzoa_orok t -> Multzoa_orok t

ebakidura_orok_fn x y = forma_normala_orok (ebakidura_orok x y)
```

forma_normala_orok funtzioak Eq t => baldintza behar duenez, bildura_orok_fn eta ebakidura_orok_fn funtzioek ere behar dute.

Adibidez,

```
ebakidura_orok_fn (Mulorok [True,False,True]) (Mulorok
[False,False,True])
```

kasuan erantzuna honako hau izango da:

```
Mulorok [False,True].
```

7.13.4. Berdintasuna, ordena eta balioak pantailan erakusteko era birdefinitzeko era data eta newtype mekanismoen bidez definitutako datu-motetan

newtype eta data mekanismoak erabiliz datu-mota berri bat definitzen dugunean, berdintasuna (==), ordena (<, <=, >, >=) eta balioak pantailan erakusteko era automatikoki ezarri ditzakegu. Hoerretarako, nahikoa da deriving (Eq, Ord, Show) eranstearekin. Bertsio automatiko horretan, datu-motako balioak karaktere-kateak balira bezala tratatzen dira. Ondorioz, ordena ohiko ordena alfabetikoa ASCII sinboloetara hedatuz finkatzen da. Kasu askotan bertsio automatiko hori egokia da, baina kasu batzuetan guk nahi dugun berdintza edo ordena edo balioak pantailan erakusteko era ez dator bat bertsio automatikoak ezartzen duenarekin.

Orain AA, BB eta CC balioek osatzen duten Hiru datu-mota definituko dugu.

```
data Hiru = AA | BB | CC
 deriving (Eq, Ord, Show, Read, Enum, Bounded)
```

Definizio hori kontuan hartuz, AA, BB eta CC balioak beraien artean desberdinak dira eta beraien arteko ordena honako hau da: AA < BB < CC. Gainera, hiru balio horiek pantailan AA, BB eta CC eran aurkeztuko dira. Baina demagun berdintasuna, ordena eta balioak pantailan aurkezteko era aldatu nahi ditugula. Hori nola egiten den azaltzeko Hiru2 mota definituko dugu eta mota horretan berdintasuna, ordena eta balioak aurkezteko era automatikoki ezarri beharrean, geuk birdefinituko ditugu:

```
data Hiru2 = AA2 | BB2 | CC2
 deriving (Read, Enum, Bounded)

instance Eq Hiru2 where
 AA2 == AA2          = True
 AA2 == x            = False
 BB2 == AA2          = False
 BB2 == x            = True
 CC2 == AA2          = False
 CC2 == x            = True

instance Ord Hiru2 where
 AA2 <= AA2          = True
 AA2 <= x            = False
 BB2 <= x            = True
 CC2 <= x            = True

instance Show Hiru2 where
 show AA2 = "Bat"
 show BB2 = "Bi"
 show CC2 = "Hiru"
```

Berdintasuna birdefinitzeko, berdintasunari dagozkion ekuazioak eman behar dira. Hiru2 datu-motaren kasuan, ekuazioak kontuan hartuz, AA2 balioa AA2 balioaren berdina da eta ez da beste bi balioen berdina. Bestalde, BB2 ez da AA2 balioaren berdina baina Hiru2 motako beste balio guztien berdina da. Beraz, BB2 eta CC2 berdinak dira. Azkenik, CC2 ez da AA2 balioaren berdina baina Hiru2 datu-motako

beste balio guztien berdina da. Laburbilduz, BB2 eta CC2 berdinak dira eta AA2 desberdina da BB2 eta CC2 balioekiko.

Elementuen arteko ordena aldatzeko, <= erlazioa definitu behar da, hau da, txikiago edo berdin erlazioa definitu behar da. Baina aurretik berdintza erlazioak definituta egon behar du. Hiru2 datu-motaren kasuan eman diren ekuazioen arabera, AA2 balioa AA2 baino txikiagoa edo berdina da baina ez da datu-motako beste edozein baino txikiagoa edo berdina. Ondorioz, AA2 balioa BB2 baino handiagoa eta CC2 baino handiagoa da. Bestalde, BB2 balioa Hiru2 motako edozein balio baino txikiagoa edo berdina da eta CC2 balioarekin gauza bera gertatzen da.

Hiru2 motakoa balioak pantailan aurkezteko era aldatzeko, show funtzioa birdefinitu behar da (funtzio hori sarrera/irteera atalean landuko da). Hiru2 motari dagokion adibide honetan eman diren ekuazioak kontuan hartuz, AA2 balioa aurkeztean "AA2" karaktere-katea aurkeztu beharrean "Bat" katea aurkeztuko da, BB2 balioa aurkeztean "BB2" karaktere-katea aurkeztu beharrean "Bi" katea aurkeztuko da eta CC2 aurkeztean "CC2" karaktere-katea aurkeztu beharrean "Hiru" katea aurkeztuko da.

Hiru eta Hiru2 datu-moten bidez, berdintza, ordena eta balioak pantailan aurkezteko era birdefinitzeko zer egin behar den erakutsi da. Hala ere, datu-mota horiek oso sinpleak dira eta erabilgarritasun gutxi dute. Baina, hor ikusi dena adibide erabilgarriagoetan ere erraz aplika dezakegu. Demagun inprimagailuetako tinta-kartutxoen koloreez osatutako *Tinta* datu-mota definitu nahi dugula. Marka komertzial askotan lau koloretako kartutxoak egon ohi dira: Horia, zian, magenta eta beltza. Koloreen izen ofizialak lau horiek izan arren, erabiltzaile edo bezero askok zian koloreari urdina esaten diote eta magenta koloreari arrosa esaten diote. Beraz, definitu nahi dugun Tinta datu-mota horretan ondo legoke lau izen ofizialez gain urdina eta arrosa ere ipintzea. Baina urdina eta zian koloreak berdinak direla garbi uztea nahi dugu eta gauza bera gertatzen da arrosa eta magentarekin:

```
data Tinta = Horia | Zian | Urdina | Magenta | Arrosa | Beltza
 deriving (Eq, Ord, Show, Read, Enum, Bounded)
```

Tinta izeneko datu-mota horretan berdintza, ordena eta balioak pantailan aurkezteko era automatikoki ezarri dira deriving klausula erabiliz eta, ondorioz, Tinta mota osatzen duten sei koloreak desberdinak dira. Zian eta Urdina balioak berdinak izatea nahi badugu eta Magenta eta Arrosa berdinak izatea nahi badugu, berdintasunaren erlazioa birdefinitu egin behar da. Berdintasunaz gain ordena eta pantailan aurkezteko era ere birdefinitu egingo ditugu. Hori dena nola egiten den erakusteko, Tinta2 izeneko datu-mota definituko dugu:

```
data Tinta2 = Horia2 | Zian2 | Urdina2 | Magenta2 | Arrosa2 | Beltza2
 deriving (Read, Enum, Bounded)

instance Eq Tinta2 where
 Horia2 == Horia2 = True
 Horia2 == x = False
 Urdina2 == Urdina2 = True
 Urdina2 == Zian2 = True
```

```
 Urdina2 == x = False
 Zian2 == Zian2 = True
 Zian2 == Urdina2 = True
 Zian2 == x = False
 Magenta2 == Magenta2 = True
 Magenta2 == Arrosa2 = True
 Magenta2 == x = False
 Arrosa2 == Arrosa2 = True
 Arrosa2 == Magenta2 = True
 Arrosa2 == x = False
 Beltza2 == Beltza2 = True
 Beltza2 == x = False

instance Ord Tinta2 where
 Horia2 <= x = True
 Zian2 <= Horia2 = False
 Zian2 <= x = True
 Urdina2 <= Horia2 = False
 Urdina2 <= x = True
 Magenta2 <= Horia2 = False
 Magenta2 <= Zian2 = False
 Magenta2 <= Urdina2 = False
 Magenta2 <= x = True
 Arrosa2 <= Horia2 = False
 Arrosa2 <= Zian2 = False
 Arrosa2 <= Urdina2 = False
 Arrosa2 <= x = True
 Beltza2 <= Beltza2 = True
 Beltza2 <= x = False

instance Show Tinta2 where
 show Horia2 = "Horia"
 show Zian2 = "Zian"
 show Urdina2 = "Zian"
 show Magenta2 = "Magenta"
 show Arrosa2 = "Magenta"
 show Beltza2 = "Beltza"
```

Berdintasuna birdefinitu izan bagenu eta ordena era automatikoan ezarri izan bagenu, Urdina2 == Zian2 berdintzarentzat erantzuna True izango litzateke eta Zian2 < Urdina2 konparazioarentzat ere erantzuna True izango litzateke. Zian2 < Urdina2 kasuan erantzuna False izateko ordena berdefinitu egin behar izan da, Hiru2 datu-motan egin den antzera. Bestalde Tinta2 datu-motan Urdina2 eta Arrosa2 balioak onartu arren, balio horiek aurkeztean Zian eta Magenta testua agertzea nahi dugu. Gainera beste balioak ere bukaerako 2 zenbakirik gabe aurkeztea nahi dugu. Hori lortzeko show funtzioa ere birdefinitu egin da.

Berdintasuna, ordena eta elementuak pantailan aurkezteko era aldatzearen komenigarritasuna Multzoa_orok t datu-motaren kasuan oraindik hobeto ikusten da. Matematikoki, multzoetan errepikapenek eta elementuen ordenak ez dute eraginik. Hori dela-eta, Mulorok [4, 8, 5, 12, 5] eta Mulorok [12, 4, 4, 5, 8] multzoak berdinak dira. Baina berdintasuna deriving klausula erabiliz era automatikoan ezartzen badugu, bi multzo horiek desberdinak izango dira. Bi multzo horiek multzo berdintzat hartzeko, berdintasuna birdefinitu egin beharko da. Bestalde, multzo bat pantailan aurkeztean elementuak giltza artean agertzea nahi dugu. Adibidez, Mulorok [4, 8, 5, 12, 5] multzoa pantailan {4, 8, 5, 12, 5} eran agertzea nahi dugu. Horretarako, show funtzioa birdefinitu beharko da. Multzoen kasuan berdintasuna, ordena eta elementuak pantailan aurkezteko era nola birdefinitu daitezkeen erakusteko, Multzoa_orok2 datu-mota definituko dugu. Datu-mota hori Multzoa_orok datu-motaren antzekoa da baina berdintasuna, ordena eta elementuak aurkezteko era automatikoki ezarri beharrean geuk birdefinituta egongo dira:

```
newtype Multzoa_orok2 t = Mulorok2 [t]
 deriving (Read)

instance Eq t => Eq (Multzoa_orok2 t) where
  s == r = ((azpimultzoa_orok2 s r) && (azpimultzoa_orok2 r s))

instance Eq t => Ord (Multzoa_orok2 t) where
  s <= r = (azpimultzoa_orok2 s r)

instance (Eq t, Show t) => Show (Multzoa_orok2 t) where
 show (Mulorok2 s) = ("{" ++ (elementu_katea s) ++ "}")

---------------------------------------------------------------------
azpimultzoa_orok2 :: Eq t => Multzoa_orok2 t -> Multzoa_orok2 t -> Bool
azpimultzoa_orok2 (Mulorok2 []) y = True
azpimultzoa_orok2 (Mulorok2 (x:s)) (Mulorok2 r)
 | x `notElem` r = False
 | otherwise = azpimultzoa_orok2 (Mulorok2 s) (Mulorok2 r)

---------------------------------------------------------------------
elementu_katea :: (Eq t,Show t) => [t] -> String

elementu_katea [] = ""
elementu_katea (x:s)
 | s == [] = show x
 | otherwise = (show x) ++ ", " ++ (elementu_katea s)
```

Bi funtzio laguntzaile erabili dira. Alde batetik, azpimultzoa_orok2 funtzioa erabili da. Funtzio horrek, multzo bat beste multzo baten azpimultzoa al den erabakiko du. Horretarako, lehenengo multzoko elementu bakoitza bigarren multzoan agertzen al den aztertuko da, ordena eta errepikapen kopurua kontuan hartu gabe. Funtzio hori kontuan hartuz, bi multzo berdinak izango dira multzo horietako bakoitza beste multzoaren azpimultzoa baldin bada. Eta multzo bat beste bat baino txikiagoa edo berdina izango da beste multzo horren azpimultzoa baldin bada. Bestalde, show funtzioa birdefinitzeko

elementu_katea funtzio laguntzailea definitu da. Funtzio horrek, pantailan erakusgarria den edozein t motako zerrenda bat emanda, zerrenda horretako elementuekin karaktere-kate bat osatzen du. Karaktere-kate hori '{' eta '}' karaktereen artean aurkeztuko du show funtzioak. Multzoak forma normalean aurkeztu nahi badira, hau da, errepikapenik gabe, forma normala kalkulatzen duen funtzioa definitu beharko genuke:

```
forma_normala_orok2 :: Eq t => Multzoa_orok2 t -> Multzoa_orok2 t

forma_normala_orok2 (Mulorok2 []) = Mulorok2 []
forma_normala_orok2 (Mulorok2 (x:s))
 | x `elem` s            = forma_normala_orok2 (Mulorok2 s)
 | x `notElem`s          = Mulorok2 (x: w)
                           where (Mulorok2 w) = (forma_normala_orok2 (Mulorok2 s))
```

Adibidez,

```
forma_normala_orok2 (Mulorok2 [6,7,3,7,8,3])
```

kasuan, honako hau lortuko genuke pantailan:

```
{6, 7, 8, 3}.
```

7.13.5. Eragiketa aritmetikoak birdefinitzeko era data eta newtype mekanismoen bidez definitutako zenbakizko datu-motetan

Lehenago, zenbaki arrunten Nat datu-mota definitu dugu:

```
data Nat = Zero | H Nat
  deriving (Eq, Ord, Show, Read)
```

Baina Haskell-ek ez daki mota hori zenbakizko datu-mota bat denik. Odorioz, H (H Zero) + H (H (H Zero)) eragiketa planteatzen badugu, emaitza gisa H (H (H (H (H Zero)))) lortu beharrean errore-mezu bat lortuko dugu. Arazo hori dela-eta, Nat motako zenbakien batura kalkulatzen duen batura_nat izeneko funtzioa definitu dugu. Baina posible da zenbaki arrunten mota beste era batera definitzea eta Haskell lengoaiak mota hori zenbakizko mota gisa onartzea. Horretarako, Num klasearen instantzia bat sortu behar da zenbaki arruntentzat. Instantzia horretan +, - y * (batuketa, kenketa eta biderketa) eragiketak definitu beharko dira. Gainera, *negate* (zeinu aldaketa), *abs* (balio absolutua), *signum* (zeinua zein den erantzuten duen funtzioa) eta *fromInteger* (Integer motako balio batetik zenbaki arrunten motarako eraldaketa nola egiten den adierazten duen funtzioa) ere defini daitezke. Negate funtzioak zeinua aldatzeko balio du (positibotik negatibora edo negatibotik positibora). Signum funtzioak zenbaki baten zeinua zein den jakiteko balio du. Adibidez, Integer motako zenbakien kasuan, signum funtzioak 1 balioa itzultzen du zenbaki positiboentzat, 0 itzultzen du zero

zenbakiarentzat eta -1 itzultzen du zenbaki negatiboentzat. Definizio hauek eransteko, zenbaki arruntentzat datu-mota berri bat definituko dugu:

```
data Nat2 = Zero2 | H2 Nat2
  deriving (Eq, Ord, Read)

instance Num Nat2 where
 Zero2 + x = x
 (H2 y) + x = (H2 (y + x))

 Zero2 - Zero2 = Zero2
 Zero2 - (H2 x) = Zero2
 (H2 x) - Zero2 = (H2 x)
 (H2 x) - (H2 y) = (x - y)

 Zero2 * x = Zero2
 (H2 x) * Zero2 = Zero2
 (H2 x) * (H2 y) = (H2 x) + ((H2 x) * y)

 negate x = error "Zenbaki negatiboak ez dira zenbaki arruntak."

 abs x = x

 signum x = 1

 fromInteger x
  | x < 0 = error "Zenbaki negatiboak ez dira zenbaki arruntak."
  | x == 0 = Zero2
  | otherwise = H2 (fromInteger (x - 1))
```

Orain zuzenean `(H2 (H2 Zero2)) + (H2 Zero2)` idatz dezakegu eta erantzuna `H2 (H2 (H2 Zero2))` izango da. Hala eta guztiz ere, zenbaki arruntzak idazteko era ez da praktikoa. Esate baterako, 100 zenbakia adierazteko H2 ehun aldiz idatzi beharko da eta gero Zero2, beharrezkoak diren berrehun parentesiekin. fromInteger funtzioak hori pixka bat erraztu egiten du. facilita un poco las cosas. 50 eta 100 zenbaki arrunten batura kalkulatu nahi badugu, honako hau idatz dezakegu:

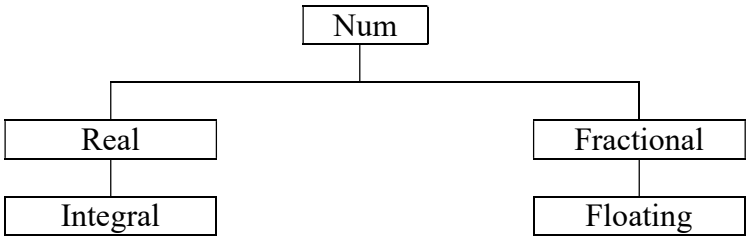
```
((fromInteger 50)::Nat2) + ((fromInteger 100)::Nat2)
```

Oraindik ere ez da oso sinplea, baina asko hobetu da zenbaki arruntak adierazteko era. Espresio horretan, Integer motako 50 eta 100 balioak Nat2 motako balioetara eraldatu edo bihurtu nahi direla adierazten da. Nat2-ra bihurtutakoan berrogeita hamar H2 eta Zero bat eta ehun H2 eta Zero bat edukiko dira eta kasu bakoitzean beharrezkoak diren parentesi guztiekin. Emaitzak ehun eta berrogeita hamar H2 eta gero Zero izango ditu (eta hirurehun parentesi). Pantailan 150 ikusi ahal izateko, Show-ren instantzia bat sortu beharko da Nat2 motarentzat:

```
instance Show Nat2 where
```

```
 show x = show (arruntetik_osora x)

arruntetik_osora :: Nat2 -> Integer
arruntetik_osora Zero2 = 0
arruntetik_osora (H2 x) = 1 + (arruntetik_osora x)
```

Bestalde, Nat2 motarentzat div eta mod eragiketak definitzeko Integral klasearen instantzia bat sortu beharko genuke eta erro karratua (sqrt) edo berreketa (**) definitzeko Floating klasearen instantzia bat sortu beharko genuke. Zenbakizko datu-moten hierarkia honako hau da:



7.13.6. Datu-moten inportazioa eta esportazioa moduluen artean

Beste modulu batean definituta dauden funtzioak inportatu daitezkeen era berean eta esportazioa ere kontrolatu daitekeen era berean, beste modulu batean definituta dauden datu-motak ere inportatu daitezke eta datu-moten esportazioa ere kontrolatu daiteke.

- **Esportazioa**ri dagokionez, esan dugu aukera desberdinak daudela **funtzioentzat**:

  - Moduluko funtzio denak esportatzeko baimena ematea. Kasu horretan modulua honela definitu behar da:

    module Moduluaren-izena where

  - Moduluko funtzio batzuk bakarrik esportatzeko baimena ematea. Kasu horretan modulua honela definitu behar da:

    module Moduluaren-izena (esportatuak izateko baimena duten funtzioen izenak) where

  Funtzioen definizioez gain **datu-mota** batzuen definizioak ere baditugunean, antzeko aukerak ditugu **esportazioa**ri dagokionez:

  - Moduluko datu-mota denak eta funtzio denak esportatzeko baimena ematea. Kasu horretan modulua honela definitu behar da:

module Moduluaren-izena where

- Moduluko datu-mota batzuk bakarrik eta funtzio batzuk bakarrik esportatzeko baimena ematea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena (esportatuak izateko baimena duten datu-moten izenak(esportatuak izateko baimena duten funtzio eraikitzaileen izenak), esportatuak izateko baimena duten funtzioen izenak) where

Datu-mota bakoitzaren izenaren ondoren, parentesien artean, esportatuak izateko baimena duten funtzio eraikitzaileak ipini behar dira.

Adibidez, lehenago ikusi dugun *Hiru* izeneko datu-mota Motak.hs moduluan baldin badago, Motak moduluaren hasiera era desberdinetakoa izan daiteke:

- *module Motak where*

  Kasu horretan moduluko datu-mota eta funtzio guztiek dute esportatuak izateko baimena.

- *module Motak (Hiru(), esportatuak izateko baimena duten beste mota eta funtzio batzuk) where*

  Kasu horretan Hiru motak esportatua izateko baimena du baina AA, BB eta CC bere funtzio eraikitzaileek ez dute baimenik, ezin dira esportatu. Beraz, beste modulu batetik Motak modulua inportatzen bada, modulu horretan *Hiru* mota erabil daiteke baina AA, BB eta CC funtzio eraikitzaileak ezingo dira erabili.

- *module Motak (Hiru(AA), esportatuak izateko baimena duten beste mota eta funtzio batzuk) where*

  Kasu horretan *Hiru* motak esportatua izateko baimena du eta AA funtzio eraikitzaileak ere bai, baina BB eta CC funtzio eraikitzaileek ez dute baimenik, ezin dira esportatu. Beraz, beste modulu batetik Motak modulua inportatzen bada, modulu horretan *Hiru* mota eta AA funtzio eraikitzailea erabil daitezke baina BB eta CC funtzio eraikitzaileak ez.

- *module Motak (Hiru(AA,BB,CC), esportatuak izateko baimena duten beste mota eta funtzio batzuk) where*

  Kasu horretan Hiru motak esportatua izateko baimena du eta AA, BB eta CC bere hiru funtzio eraikitzaileek ere bai. Beraz, beste modulu batetik Motak modulua inportatzen bada, modulu horretan Hiru izeneko mota eta AA, BB eta CC funtzio eraikitzaileak erabili ahal izango dira.

- **Inportazioa**ri dagokionez, esan dugu aukera desberdinak daudela funtzioentzat:

- Modulu batean inportagarriak diren funtzio denak inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena

- Moduluko funtzio inportagarrietatik batzuk bakarrik inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena (inportatu nahi diren funtzioen izenak)

- Moduluko funtzio inportagarri batzuk ez inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena hiding (inportatu nahi ez diren funtzioen izenak)

Funtzioen definizioez gain **datu-mota** batzuen definizioak ere baditugunean, antzeko aukerak ditugu **inportazioa**ri dagokionez:

- Moduluko datu-mota inportagarri denak eta funtzio inportagarri denak inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena

- Moduluko datu-mota eta funtzio inportagarrietatik batzuk bakarrik inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena (inportatu nahi diren datu-moten izenak(inportatu nahi diren eragiketa eraikitzaileen izenak), inportatu nahi diren funtzioen izenak)

  Datu-mota bakoitzaren izenaren ondoren, parentesien artean, inportatu nahi diren funtzio eraikitzaileak ipini behar dira.

- Moduluko datu-mota inportagarri batzuk eta funtzio inportagarri batzuk ez inportatzea. Kasu horretan modulua honela definitu behar da:

  module Moduluaren-izena where
  import Beste-moduluaren izena hiding (inportatu nahi ez diren datu-moten, funtzio eraikitzaileen eta bestelako funtzioen izenak)

Adibidez, lehenago ikusi dugun *Hiru* izeneko datu-mota Motak.hs moduluan baldin badago, eta Motak2 modulutik modulu hori inportatu nahi bada, honako aukera hauek ditugu:

- *module Motak2 where*
  import Motak

  Kasu horretan Motak moduluko datu-mota eta funtzio guztiak inportatuko dira.

- *module Motak2 where*
  import Motak (Hiru(), inportatu nahi diren beste mota eta funtzio batzuk)

  Kasu horretan Hiru mota inportatuko da baina AA, BB eta CC bere funtzio eraikitzaileak ez. Beraz, Motak2 moduluan Hiru mota erabil daiteke baina AA, BB eta CC funtzio eraikitzaileak ez.

- *module Motak2 where*
  module Motak (Hiru(AA), inportatu nahi diren beste mota eta funtzio batzuk)

  Kasu horretan Hiru mota inportatuko da eta AA funtzio eraikitzailea ere bai, baina BB eta CC funtzio eraikitzaileak ez. Beraz, Motak2 moduluan Hiru mota eta AA funtzio eraikitzailea erabil daitezke baina BB eta CC funtzio eraikitzaileak ez.

- *module Motak2 where*
  import Motak (Hiru(AA,BB,CC), inportatu nahi diren beste mota eta funtzio batzuk) where

  Kasu horretan Hiru mota inportatuko da eta AA, BB eta CC bere hiru funtzio eraikitzaileak ere bai. Beraz, Motak2 moduluan Hiru izeneko mota eta AA, BB eta CC funtzio eraikitzaileak erabili ahal izango dira.

- *module Motak2 where*
  import Motak hiding (CC, ezkutatu nahi diren beste mota eta funtzio batzuk)

  Kasu horretan Hiru mota inportatuko da eta AA eta BB funtzio eraikitzaileak ere bai, baina CC funtzio eraikitzailea ez. Beraz, Motak2 moduluan Hiru izeneko mota eta AA eta BB funtzio eraikitzaileak erabili ahal izango dira baina CC funtzio eraikitzailea ez.

- *module Motak2 where*
  import Motak hiding (Hiru, ezkutatu nahi diren beste mota eta funtzio batzuk)

  Kasu horretan Hiru mota ez da inportatuko. Beraz, Motak2 moduluan Hiru izeneko mota ezingo da erabili eta bere funtzio eraikitzaileak ere ez.

7.14. Sarrera/Irteera

Haskell lengoaiako Sarrera/Irteera-ren oinarrizko ezaugarriak aurkeztuko dira atal honetan.

7.14.1. Irteera: putStr

Mezuak pantailan aurkezteko, aurredefinitutako putStr funtzioa daukagu. Bere mota honako hau da:

```
putStr:: String -> IO ()
```

Beraz, String motako elementu bat edo karaktere-kate bat emanda, IO () motako zerbait itzuliko du. IO () motako zerbait lortuko dela esatean, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Erabiltzeko era:

```
putStr "348"
putStr "Azaroa"
```

1. adibidea:

putStr funtzioa erabiltzen duen *kaixo* izeneko funtzioaren definizioa dator jarraian. *kaixo* funtzioak beti mezu bera aurkeztuko du.: *Kaixo mundua!!*

```
kaixo:: IO ()

kaixo = putStr "Kaixo mundua!!"
```

kaixo funtzioa erabiltzeko era:

```
kaixo
```

2. adibidea:

Argumentu gisa emandako karaktere-katea aurkeztuko duen *aurkeztu* izeneko funtzioa definituko da jarraian. Funtzio honek putStr funtzioak egiten duen gauza bera egingo du. Beraz, ez du ezer berririk egiten eta putStr funtzioari izena aldatzeko bakarrik balio du.

```
aurkeztu:: String -> IO ()

aurkeztu kat = putStr kat
```

Funtzio hau erabiltzeko era:

```
aurkeztu "Azaroa"
aurkeztu "348"
```

```
aurkeztu ""
aurkeztu "2018ko azaroa"
```

7.14.2. Irteera lerroz aldatuz: putStrLn

Argumentu gisa emandako karaktere-kate bat aurkeztu eta gero lerroz aldatzeko, aurredefinitutako putStrLn funtzioa daukagu. Bere mota honako hau da:

```
putStrLn:: String -> IO ()
```

IO () motako zerbait lortuko dela esatean, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Erabiltzeko era:

```
putStrLn "Azaroa"
putStrLn "348"
putStrLn ""
putStrLn "2018ko azaroa"
```

3. adibidea:

Jarraian, argumentu gisa emandako karaktere-katea aurkeztu eta hurrengo lerrora jauzia egiten duen *aurkeztu_eta_jauzi* izeneko funtzioa definituko da. Funtzio honek putStrLn funtzioak egiten duen gauza bera egingo du. Beraz, ez du ezer berririk egiten eta putStrLn funtzioari izena aldatzeko bakarrik balio du.

```
aurkeztu_eta_jauzi:: String -> IO ()

aurkeztu_eta_jauzi kat = putStrLn kat
```

Funtzio hau erabiltzeko era:

```
aurkeztu_eta_jauzi "Azaroa"
aurkeztu_eta_jauzi "348"
aurkeztu_eta_jauzi ""
aurkeztu_eta_jauzi "2018ko azaroa"
```

7.14.3. Irteera lerroz aldatuz: "\n"

Lerro-aldaketa edo lerro-jauzia "\n" espresioaren bidez adieraz daiteke. "2018ko azaroa"++"\n" ++ "Bilbao" katea aurkezten badugu, "2018ko azaroa" eta "Bilbao" karaktere-kateak lerro desberdinetan agertuko dira.

```
aurkeztu "2018ko azaroa"++"\n" ++ "Bilbao"
```

7.14.4. do egitura: agindu-segidak idazteko

Orain "2018ko azaroa" eta "Bilbao" karaktere-kateak bi lerrotan aurkezteko *do* notazioa eta putStrLn funtzioa erabiliko ditugu. Horretarako, *ab* izeneko funtzioa definituko dugu. Funtzio horrek beti "2018ko azaroa" eta "Bilbao" karaktere-kateak bi lerrotan aurkeztuko ditu.

```
ab:: IO ()
ab = do
        putStrLn "2018ko azaroa"
        putStr "Bilbao"
```

IO () motako zerbait lortuko dela esatean, sarrera/irteerako ekintzaren bat burutuko dela baina emaitzarik ez dela gordeko adierazten da.

Sarrera/irteerarekin zerikusia duten ekintzak daudenean, ekintza horiek segida eran ipintzeko **do** egitura erabiltzen da.

7.14.5. Irteera: show (String motakoak ez diren datuak aurkezteko)

putStr eta putstrLn funtzioek karaktere-kateak aurkezteko balio dute bakarrik. Ondorioz, putStr "348" idazten badugu, 348 aurkeztuko da pantailan, baina putStr 348 idazten badugu, errorea sortuko da. Zer egin dezakegu orduan zenbakizko balio bat aurkezteko? Aurkeztu daitezkeen motetako elementuak karaktere-kate bihurtzeko balio duen *show* funtzio aurredefinitua dauka Haskell-ek. show funtzioaren mota honako hau da:

```
show:: Show t => t -> String
```

Beraz, 348 balioa aurkezteko bi aukera ditugu orain:

```
        putStr "348"
edo
        putStr (show 348)
```

Beraz, show funtzioak 348 zenbakia "348" karaktere-katera bihurtzen du.

4. adibidea:

Adibide honetan show erabiltzea nahitaezkoa da:

```
batura_aurkeztu :: Int -> Int -> IO ()
batura_aurkeztu x y = putStr ((show x) ++ " + " ++ (show y) ++ " = " ++ (show (x + y)))
```

Erabiltzeko era:

```
        batura_aurkeztu 10 5
```

5. adibidea:

Adibide honetan ere show beharrezkoa da. Aurreko funtzioarekin alderatuz, mezu desberdina aurkeztuko da.

```
batura_aurkeztu2 :: Int -> Int -> IO ()
batura_aurkeztu2 x y = putStr ((show x) ++ " gehi " ++ (show y) ++ " " ++ (show (x + y)) ++ " da")
```

Erabiltzeko era:

```
batura_aurkeztu2 10 5
```

6. adibidea:

Orain *batura_aurkeztu2* funtzioak egiten duen gauza bera *do* notazioa erabiliz egingo da.

```
batura_aurkeztu3 :: Int -> Int -> IO ()
batura_aurkeztu3 x y = do
                       putStr (show x)
                       putStr " gehi "
                       putStr (show y)
                       putStr " "
                       putStr (show (x + y))
                       putStr " da"
```

Erabiltzeko era:

```
batura_aurkeztu3 10 5
```

Sarrera/irteera-ko ekintzak daudenean, ekintza horiek zein ordenetan burutu behar diren adierazteko *do* notazioa erabiltzen da.

7. adibidea:

Hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du jarraian datorren *hotela* izeneko funtzioak do notazioa erabiliz.

```
hotela :: Int -> Int -> Bool -> IO ()
hotela e_data gela_kop jatetxea = do
                                  putStr "Eraikitze-data: "
                                  putStrLn (show e_data)
                                  putStr "Gela-kopurua: "
                                  putStrLn (show gela_kop)
                                  putStr "Jatetxea badu: "
                                  putStrLn (show jatetxea)
```

Erabiltzeko era:

```
hotela 1970 108 True
```

8. adibidea:

hotela2 izeneko funtzioak *hotela* funtzioak egiten duen gauza bera egiten du, hau da, hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du pantailan do notazioa erabiliz. Baina hirugarren parametroaren kasuan "Bai" edo "Ez" aurkeztuko da True edo False aurkeztu beharrean. Horretarako **if-then-else** egitura erabiliko da. Orain arte funtzioak definitzerakoan | notazioa erabili dugu kasu desberdinak bereizteko, baina batzuetan if-then-else egitura hobea da, batez ere do notazioaren barruan.

```
hotela2 :: Int -> Int -> Bool -> IO ()
hotela2 e_data gela_kop jatetxea = do
                                   putStr "Eraikitze-data: "
                                   putStrLn (show e_data)
                                   putStr "Gela-kopurua: "
                                   putStrLn (show gela_kop)
                                   putStr "Jatetxea badu: "
                                   if jatetxea
                                        then putStrLn "Bai"
                                        else putStrLn "Ez"
```

Erabiltzeko era:

```
hotela2 1970 108 True
```

7.14.6. Irteera: print (putStrLn + show)

Aurredefinituta dagoen print funtzioak putStrLn eta show funtzioak batera erabiliz lortzen dena egiteko balio du. Bere mota honako hau da:

```
print:: Show t => t -> IO ()
```

Beraz, print erabiltzea putStrLn eta show elkarrekin erabiltzearen baliokidea da. Ondorioz,

```
putStrLn (show 108)
```

eta

```
print 108
```

gauza bera dira.

9. adibidea:

hotela3 izeneko funtzioak *hotela* eta *hotela2* funtzioek egiten duten gauza bera egiten du, hau da, hotel bati buruzko informazioa emanda (eraikitze-data, gela-kopurua eta jatetxerik ba al duen ala ez), informazio hori aurkeztuko du pantailan do

notazioa erabiliz. Baina kasu honetan datu batzuk aurkezteko print funtzioa erabiliko da.

```
hotela3 :: Int -> Int -> Bool -> IO ()
hotela3 e_data gela_kop jatetxea = do
                                    putStr "Eraikitze-data: "
                                    print e_data
                                    putStr "Gela-kopurua: "
                                    print gela_kop
                                    putStr "Jatetxea badu: "
                                    if jatetxea
                                        then putStrLn "Bai"
                                        else putStrLn "Ez"
```

Erabiltzeko era:

```
hotela3 1970 108 True
```

7.14.7. Sarrera: getLine, <- eragilea eta read eta return funtzioak

Aurredefinitutako getLine funtzioak pantailako lerro oso bat irakurtzen du karaktere-kate gisa gordez. getLine funtzioaren mota honako hau da:

```
getLine:: IO String
```

IO String motaren bidez, sarrera/irteerako ekintzaren bat burutu dela eta gero beste funtzioren batek erabili ahal izango duen String motako datu bat gorde dela adierazten da.

10. adibidea:

jaso_eta_aurkeztu funtzioak lerro batean sartzen den testua eskatuko du, jarraian testu hori jasoko du *lerroa* izena emanez eta, bukatzeko, berriro pantailan aurkeztuko du testu hori.

```
jaso_eta_aurkeztu :: IO ()
jaso_eta_aurkeztu = do putStrLn "Lerro batean testua idatzi: "
                       lerroa <- getLine
                       putStrLn ("Hau da jasotako testua: " ++ lerroa)
```

Definizio horretan **<- eragilea** agertzen da. Eragile hori ez da esleipena. Eragile horrek IO t motako elementu bat hartu eta t motako osagaia ateratzen du. Beraz, <- eragileak IO t erako elementu bati IO zatia edo bilgarria kentzen dio.

Hori dela eta, jarraian datorren definizioa ez da zuzena. Izan ere, bertan <- eragilea esleipen gisa erabiltzen da, baina <- ez da esleipena, IO bilgarria kentzeko balio du.

```
fff :: IO ()
fff = do putStrLn "AAA"
         lerroa <- "asd"
         putStrLn ("Testua hau da: " ++ lerroa)
```

Honako funtzio hau ere ez da zuzena, 4 balioak ez duelako IO bilgarria.

```
fff2 :: IO ()
fff2 = do putStrLn "AAA"
          lerroa <- 4
          putStr ("Testua hau da: ")
          print lerroa
```

Bestalde, **return** funtzioak elementu bat IO bilgarriarekin biltzeko balio du, hau da, t motako elementu bat hartuta, IO t motako elementu bat lortzeko balio du.

return funtzioaren mota honako hau da:

```
return :: t -> IO t
```

11. adibidea:

Adibide honetako ggg funtzioak return funtzioa zertarako den erakusten du.

```
ggg :: IO ()
ggg = do putStrLn "AAA"
         lerroa <- return "asd"
         putStrLn ("Testua hau da: " ++ lerroa)
```

Hor *return "asd"* zatiaren bidez **IO String** motako elementu bat sortzen da. Elementu horri <- eragilea aplikatuz berriro **IO** zatia ezabatzen da. Beraz, `lerroa <- return "asd"` exekutatu ondoren "asd" karaktere-kateari *lerroa* izena ematea lortu da. Ondorioz

```
lerroa <- return "asd"
putStrLn ("Testua hau da: " ++ lerroa)
```

eta

```
putStrLn ("Testua hau da: " ++ "asd")
```

gauza bera dira.

12. adibidea:

Adibide honetako ggg2 funtzioan, return eta Int motako datu bat agertzen dira.

```
ggg2 :: IO ()
ggg2 = do putStrLn "AAA"
          lerroa <- return 4
          putStrLn ("Testua hau da: ")
          print lerroa
```

Hor *return 4* espresioaren bidez **IO Int** motako elementua sortzen da. Jarraian, elementu horri <- eragilea aplikatuz **IO** zatia edo bilgarria ezabatzen da. Beraz, lerroa <- return 4 espresioa exekutatu ondoren, 4 zenbakiari *lerroa* izena ematea lortu dugu. Ondorioz,

```
lerroa <- return 4
putStrLn ("Testua hau da: ")
print linea
```

eta

```
putStrLn ("Testua hau da: ")
print 4
```

gauza bera dira.

Aurredefinitutako **read** funtzioak karaktere kate bat beste mota bateko elementu bihurtzen du.

13. adibidea:

Adibide honetako *lortu_zenb* funtzioak zenbaki oso bat eskuratuko du eta zenbaki hori itzuliko du emaitza gisa beste funtzioren batek erabili ahal dezan.

```
lortu_zenb :: IO Int
lortu_zenb = do z <- getLine
                return (read z :: Int)
```

Hor *z <- getLine* espresioaren bidez zenbaki bat lortuko da baina karaktere-kate gisa (esate baterako "12") eta karaktere-kate horri z izena emango zaio. Bestalde *read z :: Int* espresioaren bidez z karaktere-kate hori zenbaki oso bihurtuko da (z-ren balio "12" karaktere-katea bada, 12 zenbakia lortuko da). Bukatzeko, *return (read z :: Int)* espresioaren bidez **IO Int** motako elementua itzuliko da. Beraz, ez da 12 itzultzen, 12 balioa IO bilgarrian sartuta itzultzen da.

14. adibidea:

Adibide honetako *batu_si* funtzioak bi zenbaki oso lortuko ditu *lortu_zenb* funtzioaren bidez eta bi zenbaki horien batura aurkeztuko du pantailan.

```
batu_si :: IO ()
batu_si = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
             z1 <- lortu_zenb
             z2 <- lortu_zenb
             print (z1 + z2)
```

Hor *lortu_zenb* funtzioak IO Int erako elementuak lortuko ditu (esate baterako, 12 baina IO bilgarriarekin). Bestalde *z1 <- lortu_zenb* espresioaren bidez zenbakizko zatiari z1 izena emango zaio. Izan ere, <- eragileak IO zatia ezabatzeko balio du.

15. adibidea:

Adibide honetako *batu_si2* funtzioak bi zenbaki oso lortuko ditu *lortu_zenb* funtzioa erabili gabe eta zenbaki horien batura aurkeztuko du pantailan. Hor *lortu_zenb* funtzioa ez denez erabiltzen, <- eta getLine erabiliz zenbaki bakoitza karaktere-kate gisa lortuko da eta gero, read erabiliz, kate horiek zenbaki bihurtuko dira.

```
batu_si2 :: IO ()
batu_si2 = do putStrLn " Bi zenbaki idatzi, bakoitza lerro batean: "
              z1 <- getLine
              z2 <- getLine
              let y1 = (read z1 :: Int)
              let y2 = (read z2 :: Int)
              print (y1 + y2)
```

Beraz, *z1 <- getLine* espresioaren bidez lehenengo zenbakia karaktere-kate gisa irakurriko da (esate baterako "12") eta kate horri z1 izena emango zaio. Jarraian, *read z1 :: Int* espresioa exekutatutakoan karaktere-kate hori zenbaki oso bihurtuko da (z1-en balioa "12" bada, 12 zenbakia lortuko da). Gainera **let** erabiltzen da zenbaki oso horri y1 izena emateko.

16. adibidea:

Adibide honetako *lortu_zerrenda* funtzioak zenbaki osozko zerrenda bat eskuratuko du eta zerrenda hori itzuliko du emaitzatzat, beste funtzioren batek erabili ahal dezan.

```
lortu_zerrenda :: IO [Int]
lortu_zerrenda = do z <- getLine
                    return (read z :: [Int])
```

Hor *z <- getLine* espresioaren bidez zenbaki osozko zerrenda bat lortuko da baina karaktere-kate gisa (esate baterako "[4, 12, 6]") eta karaktere-kate horri z izena

emango zaio. Bestalde *read z :: [Int]* espresioaren bidez z karaktere-kate hori zenbaki osozko zerrenda bihurtuko da (z-ren balioa "[4, 12, 6]" karaktere-katea bada, [4, 12, 6] zenbaki osozko zerrenda lortuko da). Bukatzeko, *return (read z :: Int)* espresioaren bidez **[4, 12, 6]** elementua IO bilgarriarekin itzuliko da. Beraz, ez da [4, 12, 6] itzuliko, [4, 12, 6] IO bilgarriarekin itzuliko da.

17. adibidea:

Funtzio honek Int motako zerrenda bat lortu eta bertako zenbakien batura aurkeztuko du pantailan:

```
batu_zerrenda :: IO ()
batu_zerrenda = do putStrLn "Zerrenda bat idatzi:"
                   zer <- lortu_zerrenda
                   print (sum zer)
```

7.14.8. Sarrera/irteerako prozesuen errepikapena: errekurtsibitatea

Haskell lengoaia erabiliz datu-eskatzea bere baitan duen prozesu bat errepikatu nahi denean, errekurtsibitatea behar da. Horren erakusle, jarraian datozen adibideak dira.

18. adibidea:

batu_si2_sek funtzioak bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta jarraian prozesua behin eta berriz errepikatuko du amaierarik gabe. Programa bukatzeko, konpiladoretik edo Haskell menutik gelditu beharko da. Beraz, alde horretatik ez da oso egokia.

```
batu_si2_sek :: IO ()
batu_si2_sek = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                  z1 <- getLine
                  z2 <- getLine
                  let y1 = (read z1 :: Int)
                  let y2 = (read z2 :: Int)
                  print (y1 + y2)
                  batu_si2_sek
```

19. adibidea:

Adibide honetako *batu_si3_sek* funtzioak aurreko adibideko *batu_si2_sek* funtzioak egiten duen gauza bera egiten du baina emaitza aurkezterakoan mezu desberdina erabiltzen da. Beraz, funtzio honek bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta prozesua behin eta berriz errepikatuko du amaierarik gabe. Programa bukatzeko konpiladoretik edo Haskell menutik gelditu beharko da. Beraz, alde horretatik ez da oso egokia.

```
batu_si3_sek :: IO ()
batu_si3_sek = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                  z1 <- getLine
                  z2 <- getLine
                  let y1 = (read z1 :: Int)
                  let y2 = (read z2 :: Int)
                  putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                  print (y1 + y2)
                  batu_si3_sek
```

20. adibidea:

batu_si_sek_buk funtzioak bi zenbaki oso eskatu, jaso, batura kalkulatu eta aurkeztu eta prozesu hori guztia behin eta berriz errepikatuko du erabiltzaileak bukatzea nahi duela esan arte. Beraz, programa bukatzeko era egokiagoa da. Hala ere, erabiltzaileari jarraitzea nahi al duen galdetutakoan, erabiltzaileak b (bai) edo e (ez) ez badu erantzuten ere, e erantzun duela ulertuko da. Beraz, erantzuna ez da kontrolatzen eta alde horretatik ez da guztiz egokia.

Funtzio honetan erabiltzailearen erantzuna String motako elementu gisa tratatzen da eta horregatik "b" (komatxo bikoitzekin) erantzun al duen aztertuko da.

```
batu_si_sek_buk :: IO ()
batu_si_sek_buk = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                     z1 <- getLine
                     z2 <- getLine
                     let y1 = (read z1 :: Int)
                     let y2 = (read z2 :: Int)
                     putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                     print (y1 + y2)
                     putStrLn "Beste batuketarik? (b/e): "
                     besterik <- getLine
                     if besterik == "b"
                       then batu_si_sek_buk
                       else putStrLn "Bukaera."
```

21. adibidea:

batu_si_sek_buk2 funtzioak, *batu_si_sek_buk* funtzioak egiten duen gauza bera egiten du eta erabiltzailearen b edo e erantzunarekin arazo bera dauka. Aldaketa bakarra honako hau da: funtzio honetan erabiltzailearen erantzuna String motako elementu gisa mantendu beharrean, Char motakoa dela kontsideratzen da eta horregatik 'b' (komatxo sinpleekin edo apostrofoarekin) erantzun al duen aztertuko da.

```
batu_si_sek_buk2 :: IO ()
batu_si_sek_buk2 = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                      z1 <- getLine
                      z2 <- getLine
                      let y1 = (read z1 :: Int)
                      let y2 = (read z2 :: Int)
```

```
                        putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                        print (y1 + y2)
                        putStrLn "Beste batuketarik? (b/e): "
                        besterik <- getLine
                        let erantzuna = head besterik
                        if erantzuna == 'b'
                          then batu_si_sek_buk2
                          else putStrLn "Bukera."
```

22. adibidea:

batu_si_sek_buk3 funtzioak, *batu_si_sek_buk* eta *batu_si_sek_buk2* funtzioek egiten dutenaren antzekoa egiten du, baina hemen erabiltzaileak derrigorrez b (bai) edo e (ez) erantzun beharko du, ez da beste erantzunik onartuko.

Funtzio honen beste ezaugarri garrantzitsu bat honako hau da: Erabiltzaileak gutxienez batuketa bat burutu beharko duela.

```
batu_si_sek_buk3 :: IO ()
batu_si_sek_buk3 = do putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                      z1 <- getLine
                      z2 <- getLine
                      let y1 = (read z1 :: Int)
                      let y2 = (read z2 :: Int)
                      putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                      print (y1 + y2)
                      erantzuna <- erantzuna_eskatu
                      if erantzuna == 'b'
                        then batu_si_sek_buk3
                        else putStrLn "Bukaera."
```

Hor *erantzuna_eskatu* funtzioak erabiltzaileari b (bai) edo e (ez) erantzuteko eskatzen dio eta erabiltzaileak b edo e erantzun arte errepikatuko du eskatze-prozesua. Ez du beste erantzunik onartuko.

```
erantzuna_eskatu:: IO Char
erantzuna_eskatu = do  putStrLn "beste batuketarik? (b/e): "
                       besterik <- getLine
                       let er = head besterik
                       if er `elem` "be"
                         then (return er)
                         else do
                                 putStrLn "Erantzuna ez da egokia..."
                                 erantzuna_eskatu
```

erantzuna_eskatu funtzioan do bi aldiz erabiltzen da, aginduz osatutako azpisegida bat nola defini daitekeen erakutsiz.

23. adibidea:

batu_si_sek_buk4 funtzioak batu_si_sek_buk3 funtzioak egiten duenaren antzekoa egiten du, baina hemen erabiltzaileak bukatzea aukera dezake batuketa bat bera ere egin gabe. Horretarako, batuketarik egin nahi al duen galdetzen zaio erabiltzaileari hasieran.

```
batu_si_sek_buk4 :: IO ()
batu_si_sek_buk4 = do erantzuna <- erantzuna_eskatu2
                      if erantzuna == 'b'
                        then do
                                putStrLn "Bi zenbaki idatzi, bakoitza lerro batean:"
                                z1 <- getLine
                                z2 <- getLine
                                let y1 = (read z1 :: Int)
                                let y2 = (read z2 :: Int)
                                putStr ((show y1) ++ " + " ++ (show y2) ++ " = ")
                                print (y1 + y2)
                                batu_si_sek_buk4
                        else putStrLn "Bukaera."
```

Hor *erantzuna_eskatu2* funtzioak, erabiltzaileari b (bai) edo e (ez) erantzuteko eskatzen dio eta erabiltzaileak b edo e erantzun arte errepikatuko du eskatze-prozesua. Funtzio honek *erantzuna_eskatu* funtzioak egiten duen gauza bera egiten du baina mezu desberdina aurkezten du hasieran, funtzio desberdin batean erabiliko delako.

```
erantzuna_eskatu2:: IO Char
erantzuna_eskatu2 = do putStrLn "Bi zenbaki batzerik nahi al duzu? (b/e): "
                       besterik <- getLine
                       let er = head besterik
                       if er `elem` "be"
                         then (return er)
                         else do
                                 putStrLn "Erantzuna ez da egokia..."
                                 erantzuna_eskatu2
```

erantzuna_eskatu2 funtzioan ere, do bi aldiz erabiltzen da.

7.15. Fitxategiak

Haskell lengoaia fitxategiak erabiltzeko oinarrizko baliabideak aurkeztuko dira atal honetan.

7.15.1. Fitxategiekin aritzeko Prelude moduluan dauden funtzioak

Haskell lengoaiako oinarrizko moduluan, hau da, Prelude moduluan fitxategiekin aritzeko oinarrizko hiru funtzio daude. Funtzio horietako bat fitxategiak irakurtzeko da (readFile) eta beste biak fitxategietan idazteko (writeFile eta appendFile).

7.15.1.1. readFile funtzioa (fitxategiak irakurtzeko) eta FilePath mota

Fitxategiak irakurtzeko balio duen **readFile** funtzioak fitxategi osoa karaktere-kate bakar gisa irakurtzen du. Beraz, String motako elementu bat itzuliko du eta hor fitxategiaren eduki osoa izango dugu. Irakurri nahi den fitxategiak lerro-jauziak izan arren, lerro guztiek batera (lerro-jauziak barne) karaktere-kate bat osatuko dute readFile funtzioaren ikuspuntutik.

readFile funtzioaren mota honako hau da:

```
readFile :: FilePath -> IO String
```

FilePath mota String motaren sinonimoa da eta Prelude moduluan definituta dago. String erabili beharrean FilePath erabiltzen da funtzioari eman beharreko datua fitxategi baten izena (edo fitxategira iristeko bidea) dela argiago gera dadin.

Demagun fitxategia1.txt izeneko fitxategi bat dugula eta fitxategi horren edukia honako hau dela:

fitxategia1.txt
A aaaa B bbbb C cccc 2018

Honako g funtzio honek fitxategia1.txt fitxategiko edukia irakurri eta pantailan aurkeztuko du:

```
g :: IO()
g = do
      putStrLn "Fitxategia irakurtzen..."
      edukia <- readFile "fitxategia1.txt"
      putStrLn ("Edukia honako hau da: \n" ++ edukia)
```

readFile funtzioak IO String motako elementu bat itzuliko du eta <- eragilearen bidez IO zatia kenduko da. Horrela, 'edukia' izeneko aldagaian String motako balioa gordeko da.

Pantailan honako hau aurkeztuko da `g` funtzioa exekutatutakoan:

Pantaila

`Fitxategia irakurtzen...`
`Edukia honako hau da:`
`A aaaa B bbbb`
`C cccc`
`2018`

Beraz, irakurri nahi fitxategiaren kokapena (fitxategiaren izena barne) eman behar zaio readFile funtzioari sarrerako datu gisa. Kokapen hori karaktere-kate baten bidez adierazi beharko da. Oraintxe definitu den g funtzioaren kasuan, g-ren definizioa duen fitxategia eta fitxategia1.txt izeneko fitxategia karpeta berean daudela suposatzen da. Horregatik, nahikoa da fitxategiaren izena ematea, hau da, fitxategia1.txt.

Aldiz, g-ren definizioa duen fitxategia bere barnean duen karpetarekiko S izeneko azpikarpetan dagoen fitxategia2.txt ikarri nahi bada, readFile funtzioari datu gisa eman beharreko bidea S/fitxategia2.txt izango da. Honako g2 funtzioak S azpikarpetan dagoen fitxategia2.txt irakurriko du:

```
g2 :: IO()
g2 = do
     putStrLn "Fitxategia irakurtzen..."
     edukia <- readFile "S/fitxategia2.txt"
     putStrLn ("Edukia honako hau da: \n" ++ edukia)
```

Demagun fitxategia2.txt fitxategiaren edukia honako hau dela:

fitxategia2.txt
D dddd. E eeee FF f. 2020.

Pantailan honako hau aurkeztuko da `g2` funtzioa exekutatutakoan:

Pantaila

`Fitxategia irakurtzen...`
`Edukia honako hau da:`
`D dddd.`
`E eeee FF f.`
`2020.`

g eta g2 funtzioetan, irakurri beharreko fitxategia funtzio horien definizioan bertan agertzen da. Baina irakurri beharreko fitxategia funtzioari argumentu gisa ere eman diezaiokegu. Horrela, funtzio berak edozein fitxategirentzat balioko du.

g3 funtzioa g eta g2 funtzioen orokorpena da. g3 funtzioak irakurri beharreko fitxategiaren izenaz osatutako karaktere-katea jasoko du argumentu gisa. Fitxategiaren izena ematean, fitxategira iristeko bidea ere eman beharko da beste karpeta batean baldin badago.

```
g3 :: FilePath -> IO()  -- FilePath mota String motaren sinonimoa da
g3 fitx = do
        putStrLn "Fitxategia irakurtzen..."
        edukia <- readFile fitx
        putStrLn ("Edukia honako hau da: \n" ++ edukia)
```

`g3 "fitxategia1.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

`Fitxategia irakurtzen...`
`Edukia honako hau da:`
`A aaaa B bbbb`
`C cccc`
`2018`

`g3 "S/fitxategia2.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

`Fitxategia irakurtzen...`
`Edukia honako hau da:`
`D dddd.`
`E eeee FF f.`
`2020.`

7.15.1.2. writeFile funtzioa (fitxategietan idazteko)

Orain, fitxategi batean eduki berria idazteko balio duen **writeFile** funtzioa aztertuko dugu. writeFile funtzioa erabiltzen denean, fitxategian aurretik zegoen guztia ezabatu egingo da.

writeFile funtzioaren mota honako hau da:

```
writeFile :: FilePath -> String -> IO ()
```

Beraz, zein fitxategitan idatzi nahi den eta zer testu idatzi nahi den zehaztu beharko dira.

Demagun fitxategia3.txt fitxategia dugula eta bere edukia honako hau dela:

fitxategia3.txt

```
Haskell programazio-lengoaia
Bilbo
2018
```

h funtzioak, argumentu gisa ematen zaion fitxategiaren edukia irakurri, pantailan aurkeztu, "Haskell funtzioetan oinarritutako lengoaia da." karaktere-katea fitxategi horretan idatzi eta, bukatzeko, fitxategiaren edukia berriro aurkeztuko du:

```
h :: FilePath -> IO()
h fitx = do
        putStrLn "Fitxategia irakurtzen..."
        edukia <- readFile fitx
        putStrLn (fitx ++ " fitxategiaren hasierako edukia honako hau da: \n" ++ edukia)
        putStrLn "Fitxategia idazten..."
        writeFile fitx "Haskell funtzioetan oinarritutako\nlengoaia da."
        putStrLn "Fitxategia irakurtzen..."
        edukia <- readFile fitx
        putStrLn (fitx ++ " fitxategiaren bukaerako edukia honako hau da: \n" ++ edukia)
```

`h "fitxategia3.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

```
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren hasierako edukia honako hau da:
Haskell programazio-lengoaia
Bilbo
2018
Fitxategia idazten...
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren bukaerako edukia honako hau da:
Haskell funtzioetan oinarritutako
lengoaia da.
```

Fitxategian idatzi nahi den testua argumentu gisa eman ahal izateko, h funtzioa orokortu genezake.

h1 funtzioak, lehenengo argumentutzat ematen zaion fitxategiaren edukia irakurri, pantailan aurkeztu, bigarren argumentutzat ematen zaion karaktere-katea fitxategi horretan idatzi eta fitxategi horren eduki berria pantailan aurkeztuko du:

```
h1 :: FilePath -> String -> IO()
h1 fitx testua =
  do
    putStrLn "Fitxategia irakurtzen..."
    edukia <- readFile fitx
    putStrLn (fitx ++ " fitxategiaren hasierako edukia honako hau da: \n" ++ edukia)
    putStrLn "Fitxategia idazten..."
    writeFile fitx testua
    putStrLn "Fitxategia irakurtzen..."
```

```
  edukia <- readFile fitx
  putStrLn (fitx ++ " fitxategiaren bukaerako edukia honako hau da: \n" ++ edukia)
```

fitxategia3.txt fitxategiaren edukia honako hau baldin bada:

fitxategia3.txt
Haskell funtzioetan oinarritutako lengoaia da.

`h1 "fitxategia3.txt" "Haskell lengoaian motek garrantzi handia dute."` exekutatutakoan pantailan honako hau aurkeztuko da:

Pantaila

```
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren hasierako edukia honako hau da:
Haskell funtzioetan oinarritutako
lengoaia da.
Fitxategia idazten...
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren bukaerako edukia honako hau da:
Haskell lengoaian motek garrantzi handia dute.
```

Eta "fitxategia3.txt" fitxategiaren edukia honako hau izango da:

fichero3.txt
Haskell lengoaian motek garrantzi handia dute.

7.15.1.3. appendFile funtzioa (fitxategietan idazteko)

Orain **appendFile** funtzioa aztertuko dugu. Funtzio horrek fitxategi batean eduki berria eransteko balio du, lehendik zegoen edukia mantenduz. Eduki berria fitxategiaren bukaeran eransten da.

appendFile funtzioaren mota honako hau da:

```
appendFile :: FilePath -> String -> IO ()
```

Beraz, zein fitxategitan erantsi nahi den testua eta zer testu erantsi nahi den zehaztu behar dira.

Demagun fitxategia3.txt fitxategiaren edukia honako dela:

fichero3.txt
Haskell lengoaian motek garrantzi handia dute.

h2 funtzioak, lehenengo argumentutzat ematen zaion fitxategiaren edukia irakurri, pantailan aurkeztu, bigarren argumentutzat ematen zaion karaktere-katea fitxategi horren edukiari erantsi eta fitxategi horren eduki berria pantailan aurkeztuko du:

```
h2 :: FilePath -> String -> IO()
h2 fitx testua =
  do
    putStrLn "Fitxategia irakurtzen..."
    edukia <- readFile fitx
    putStrLn (fitx ++ " fitxategiaren hasierako edukia honako hau da: \n" ++ edukia)
    putStrLn "Fitxategiari eduki berria eransten..."
    appendFile fitx testua
    putStrLn "Fitxategia irakurtzen..."
    edukia <- readFile fitx
    putStrLn (fitx ++ " fitxategiaren bukaerako edukia honako hau da: \n" ++ edukia)
```

`h2 "fitxategia3.txt" "\nHaskell lengoaiak motak inferi ditzake."` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

```
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren hasierako edukia honako hau da:
Haskell lengoaian motek garrantzi handia dute.
Fitxategiari eduki berria eransten...
Fitxategia irakurtzen...
fitxategia3.txt fitxategiaren bukaerako edukia honako hau da:
Haskell lengoaian motek garrantzi handia dute.
Haskell lengoaiak motak inferi ditzake.
```

Hor ikus daitekeenez, hasierako edukia mantendu egin da eta testu berria erantsi da. h2 funtzioari deitzean, bigarren argumentua '\n' karakterearekin hasten den kate bat da. Hori dela-eta, fitxategia3.txt fitxategiko edukia bi lerrotan banatuta gelditzen da.

readFile, writeFile eta appendFile funtzioen oinarrizko erabileraren aurkezpenarekin bukatzeko, argumentutzat hiru fitxategi hartu eta lehenengo bi fitxategietako edukia hirugarrengoan kopiatzen duen h3 funtzioa definituko da. Hirugarren fitxategian lehendik egon zitekeen informazioa galdu egingo da.

Demagun fitxategia4.txt, fitxategia5.txt eta fitxategia6.txt fitxategiak ditugula eta fitxategi horien edukiak honako hauek direla:

fitxategia4.txt

```
A aaaaa.
B bbbb.
C ccc.
```

fitxategia5.txt

```
D dd.
E e.
```

fitxategia6.txt
000. 111.

h3 funtzioaren definizioa honako hau da:

```
h3 :: FilePath -> FilePath -> FilePath -> IO()
h3 fitx1 fitx2 fitx3 =
  do
    putStrLn "Hirugarren fitxategia irakurtzen..."
    edukia3 <- readFile fitx3
    putStrLn (fitx3 ++ " fitxategiaren hasierako edukia honako hau da: \n" ++ edukia3)
    putStrLn "Lehenengo fitxategia irakurtzen..."
    edukia1 <- readFile fitx1
    putStrLn "Hirugarren fitxategian idazten..."
    writeFile fitx3 edukia1
    putStrLn "Hirugarren fitxategia irakurtzen..."
    edukia3 <- readFile fitx3
    putStrLn (fitx3 ++ " fitxategiaren edukia une honetan honako hau da: \n" ++ edukia3)
    putStrLn "Bigarren fitxategia irakurtzen..."
    edukia2 <- readFile fitx2
    putStrLn "Hirugarren fitxategiari eduki berria eransten..."
    appendFile fitx3 ("\n" ++ edukia2)
    putStrLn "Hirugarren fitxategia irakurtzen..."
    edukia3 <- readFile fitx3
    putStrLn (fitx3 ++ " fitxategiaren bukaerako edukia honako hau da: \n" ++ edukia3)
```

`h3 "fitxategia4.txt" "fitxategia5.txt" "fitxategia6.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

```
Hirugarren fitxategia irakurtzen...
fitxategia6.txt fitxategiaren hasierako edukia honako hau da:
000.
111.
Lehenengo fitxategia irakurtzen...
Hirugarren fitxategian idazten...
Hirugarren fitxategia irakurtzen...
fitxategia6.txt fitxategiaren edukia une honetan honako hau da:
A aaaaa.
B bbbb.
C ccc.
Bigarren fitxategia irakurtzen...
Hirugarren fitxategiari eduki berria eransten...
Hirugarren fitxategia irakurtzen...
fitxategia6.txt fitxategiaren bukaerako edukia honako hau da:
A aaaaa.
B bbbb.
```

```
C ccc.
D dd.
E e.
```

7.15.2. Prelude moduluko 'lines' funtzioa

Aurreko atalean ikusi da readFile funtzioak fitxategi baten eduki osoa karaktere-kate bakar gisa eskuratzen duela. readFile funtzioak lortutako karaktere-kate horretan egon daitezkeen lerroak bereizteko 'lines' funtzioa erabil dezakegu. Funtzio hori Prelude moduluan dago. 'lines' funtzioaren mota honako hau da:

```
lines :: String -> [String]
```

Hainbat lerro izan ditzakeen karaktere-kate bat emanda, lerroak bereiztuz lortzen den zerrenda itzuliko du. Beraz, zerrenda horretako elementu bakoitza hasierako karaktere-kateko lerro bat izango da.

Orain h4 funtzioaren definizioa aurkeztuko da. Funtzio horrek, fitxategi bat emanda, fitxategi horren edukiko lehenengo hiru lerroak aurkeztuko ditu pantailan. Gainera lerro-zenbakia ere aurkeztuko du:

```
h4 :: FilePath -> IO()
h4 fitx =
    do
      putStrLn "Fitxategia irakurtzen..."
      edukia <- readFile fitx
      let lerroka = (lines edukia)
      putStrLn ("Lehenengo hiru lerroetako edukia honako hau da: \n")
      putStrLn ("-Lehenengo lerroa: \n" ++ (head lerroka))
      putStrLn ("-Bigarren lerroa: \n" ++ (head (tail lerroka)))
      putStrLn ("-Hirugarren lerroa: \n" ++ (head (tail (tail lerroka))))
```

Demagun fitxategia6.txt fitxategiaren edukia honako hau dela:

fitxategia6.txt
A aaaaa. B bbbb. C ccc. D dd. E e.

`h4 "fitxategia6.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila
`Fitxategia irakurtzen...` `Lehenengo hiru lerroetako edukia honako hau da:`

```
-Lehenengo lerroa:
A aaaaa.
-Bigarren lerroa:
B bbbb.
-Hirugarren lerroa:
C ccc.
```

Dena den, emandako fitxategiak hiru lerro baino gutxiago baldin baditu, errorea sortuko da.

Demagun fitxategia3.txt fitxategiaren edukia honako hau dela:

fitxategia3.txt
Haskell lengoaian motek garrantzi handia dute. Haskell lengoaiak motak inferi ditzake.

`h4 "fitxategia3.txt"` exekutatutakoan, pantailan honako hau aurkeztuko da:

Pantaila

```
*** Exception: Prelude.head: empty list
Fitxategia irakurtzen...
Lehenengo hiru lerroetako edukia honako hau da:

-Lehenengo lerroa:
Haskell lengoaian motek garrantzi handia dute.
-Bigarren lerroa:
Haskell lengoaiak motak inferi ditzake.
-Hirugarren lerroa:
```

h5 funtzioa h4 funtzioaren orokorpen bat da. h5 funtzioak lerro denak aurkeztuko ditu, ez hiru bakarrik. Horretarako, h6 funtzio laguntzailea erabiliko du:

```
import Data.List – aurredefinitutako genericLength funtzioa erabiltzeko
import Data.String  -- aurredefinitutako lines funtzioa erabiltzeko

h5 :: FilePath -> IO()
h5 fitx =
    do
      putStrLn "Fitxategia irakurtzen..."
      edukia <- readFile fitx
      let lerroka = (lines edukia)
      putStrLn ("Lerro bakoitzeko edukia honako hau da: \n")
      h6 lerroka 1

h6 :: [String] -> Integer -> IO()
h6 kate_zerrenda lerro_zenbakia =
    do
      if (genericLength kate_zerrenda) == 0
        then putStrLn "Hutsa"
```

```
    else if (genericLength kate_zerrenda) == 1
        then putStrLn ((show lerro_zenbakia) ++ ". lerroa" ++ ": \n" ++ (head kate_zerrenda))
        else do
            putStrLn ((show lerro_zenbakia) ++ ". lerroa" ++ ": \n" ++ (head kate_zerrenda))
            h6 (tail kate_zerrenda) (lerro_zenbakia + 1)
```

h6 funtzioak lehenengo argumentu gisa jasotzen duen zerrenda osatzen duten karaktere-kateak aurkeztuko ditu banan-banan. h5 funtzioaren ikuspuntutik, kate horiek h5 funtzioak sarrerako datu gisa jasotzen duen fitx fitxategia osatzen duten lerroak dira. Kontuan izan, genericLength funtzioa erabiltzeko Data.List mudulua inportatu behar dela (funtzio horrek zerrenda baten luzera edo osagai kopurua kalkulatzeko balio du). Bestalde, lines funtzioa erabiltzeko Data.String modulua inportatu behar da (lines funtzioak fitxategi bat osatzen duten lerroak bereizten ditu).

fitxategia6.txt fitxategiaren edukia honako hau baldin bada:

fitxategia6.txt

```
A aaaaa.
B bbbb.
C ccc.
D dd.
E e.
```

`h5 "fitxategia6.txt"`, exekutatu ondoren, pantailan honako hau aurkeztuko da:

Pantaila

```
Fitxategia irakurtzen...
Lerro bakoitzeko edukia honako hau da:

1. lerroa:
A aaaaa.
2. lerroa:
B bbbb.
3. lerroa:
C ccc.
4. lerroa:
D dd.
5. lerroa:
E e.
```

7.15.3. Fitxategiekin aritzeko System.IO moduluan dauden funtzioak

Aurreko atalean aurkeztu den readFile funtzioak fitxategi osoa aldi batean irakurtzen du. Fitxategian datu asko baldin badaude, datu horiek denak memorian mantendu beharko dira. Horrela, baliabide gehiegi erabiliko dira eta eraginkortasuna galdu egingo da. Gainera, askotan fitxategia lerroka irakurtzea nahiago izango da. Lerro bakoitza bezero

bati buruzko informazioa edo erosketa bati buruzko informazioa eta abar izan daiteke. Horregatik, ohikoena aldi bakoitzean lerro bat irakurtzea eta lerro horri dagokion eragiketa egitea izango da. Esandako hori guztia kontuan hartuz, fitxategiekiekin lanean aritzeko Haskell-ek dituen beste funtzio batzuk aurkeztuko dira atal honetan. Funtzio hauek ere guztiz oinarrizkoak dira baina aurretik aurkeztu diren readFile, writeFile eta appendFile funtzioak baino eraginkorragoak.

Atal honetan aurkeztuko diren funtzioak System.IO moduluan definituta daude. Ondorioz, modulu hori inportatu beharko da.

7.15.3.1. openFile: ReadMode, WriteMode, AppendMode eta Handle mota

Fitxategi batetik irakurtzeko edo fitxategi batean idazteko, fitxategia ireki egin beharko da eta gainera irekitzeko modua adierazi beharko da, hau da, fitxategitik irakurri egingo al den, fitxategian idatzi egingo al den ala fitxategiari datuak erantsi egingo al zaizkion zehaztu beharko da. Fitxategiko edukia irakurri nahi bada, ReadMode eran ireki beharko da. Fitxategian zerbait idatzi nahi bada, WriteMode eran ireki beharko da (modu honetan irekiz gero, fitxategian lehendik zegoen informazio dena galdu egingo da). Fitxategian dagoen informazioari eduki gehiago eranstea nahi bada, AppendMode eran ireki beharko da (modu honetan irekiz gero, fitxategian lehendik zegoen informazio dena mantendu egingo da).

Fitxategi bat irekitzeko openFile funtzioa erabili behar da. Bere mota honako hau da:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Beraz, datu gisa fitxategi baten izena (karaktere-kate bat) eta fitxategia irekitzeko era behar ditu (ReadMode, WriteMode edo AppendMode). openFile funtzioak fitxategiko datuak atzitezko eta fitxategia maneiatzeko balio duen elementu bat itzuliko du. Elementu horri *fitxagirako sarbidea* deituko diogu. Horrela, fitxategi fisikoa eta fitxategi fisiko hori ordezkatzen duen elementua garbi bereizita gelditzuko dira. Fitxategia ireki ondoren, fitxategia bera erabili beharrean fitxategirako sarbidea erabiliko da. Fitxategirako sarbidearen mota Handle da. openFile funtzioak IO Handle motako erantzuna itzultzen duenez, <- eragilea erabiliko da IO egitura kentzeko Handle motakoa den zatiarekin gelditzeko.

7.15.3.2. hGetLine, hClose eta xehetasun gehiago Handle motari buruz

Fitxategi batetik lerro bat irakurtzeko, hGetLine funtzioa daukagu. Funtzio horren mota honako hau da:

```
hGetLine :: Handle -> IO String
```

Irakurri nahi den fitxategiari dagokion sarbidea emanda, IO String motako emaitza itzuliko du. IO zatia kentzeko, <- eragilea erabiliko da eta gelditzen den String motako elementua fitxategiko lerro bat izango da.

Fitxategiarekin egin beharreko erakiketak bukatutakoan fitxategia itxi egin behar da. Horretarako hClose funtzioa daukagu. Funtzio horren mota honako hau da:

```
hClose :: Handle -> IO ()
```

hClose funtzioak datu gisa fitxategi baterako sarbidea jasoko du eta sarbide horri dagokion fitxategia itxi egingo du.

Orain, datu gisa emandako fitxategiko lehenengo hiru lerroak irakurri eta pantailan aurkeztuko dituen i izeneko funtzioa definituko da. Hasteko, fitxategia ireki eta fitxategirako sarbidea sortu beharko dira. Bukaeran, fitxategia itxi egin beharko da sarbidea erabiliz.

```
i :: FilePath -> IO()
i fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      lerroa <- hGetLine sarbidea
      putStrLn ("Lehenengo lerroa: " ++ lerroa)
      lerroa <- hGetLine sarbidea
      putStrLn ("Bigarren lerroa: " ++ lerroa)
      lerroa <- hGetLine sarbidea
      putStrLn ("Hirugarren lerroa: " ++ lerroa)
      putStrLn "Fitxategia ixten..."
      hClose sarbidea
```

Definizioan ikus daiteke i funtzioan fitx parametroaren bidez adierazitako fitxategia ireki denean, fitxategi horretarako sarbideari 'sarbidea' izena eman zaiola. Hortik aurrera, fitx erabili beharrean 'sarbidea' izena erabili da fitxategiaren gainean eragiketak egiteko. Adibide honetan bakarrik irakurketak egiten dira. Hain zuzen ere, fitxategiko lehenengo hiru lerroak irakurtzen dira hGetLine erabiliz. hGetLine funtzioak IO String motako erantzuna itzultzen duenez, <- eragilea erabili behar da 'lerroa' izeneko aldagaian String motako zatia gordetzeko.

Demagun fitxategia10.txt fitxategiak honako eduki hau duela:

fitxategia10.txt
40 100 20 700 50

`i "fitxategia10.txt"` exekutatutakoan, pantailan honako hau ikusiko da:

Pantaila
`Fitxategia irekitzen...` `Lehenengo lerroa: 40`

```
Bigarren lerroa: 100
Hirugarren lerroa: 20
Fitxategia ixten...
```

Handle mota agertu zaigunez, mota horretako elementuak nolakoak diren ikusiko dugu orain. Horretarako, i2 izeneko funtzioa definituko dugu eta funtzio horren bidez 'sarbidea' aldagaiaren balioa aurkeztuko da pantailan. Horretaz aparte, i2 funtzioak i funtzioak egiten duen gauza bera egingo du:

```
i2 :: FilePath -> IO()
i2 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      putStrLn (show sarbidea) -- Handle motako elementua aurkezteko
      lerroa <- hGetLine sarbidea
      putStrLn ("Lehenengo lerroa: " ++ lerroa)
      lerroa <- hGetLine sarbidea
      putStrLn ("Bigarren lerroa: " ++ lerroa)
      lerroa <- hGetLine sarbidea
      putStrLn ("Hirugarren lerroa: " ++ lerroa)
      putStrLn "Fitxategia ixten..."
      hClose sarbidea
```

`i2 "fitxategia10.txt"` exekutatutakoan, pantailan honako hau ikusiko da:

Pantaila

```
Fitxategia irekitzen...
{handle: fitxategia10.txt}
Lehenengo lerroa: 40
Bigarren lerroa: 100
Hirugarren lerroa: 20
Fitxategia ixten...
```

Hor ikus daiteke Handle motakoa den 'sarbidea' aldagaiaren balioa honako hau dela:

```
{handle: fitxategia10.txt}
```

Berriro i funtziora itzuliz, i funtzioari hiru lerro baino gutxiago dituen fitxategi bat ematen badiogu, errorea sortuko da. Adibidez, fitxategia11.txt fitxategiaren edukia honako hau baldin bada:

fitxategia11.txt

```
8000
```

`i "fitxategia11.txt"` exekutatutakoan, pantailan honako hau ikusiko da:

Pantaila

```
*** Exception: fitxategia11.txt: hGetLine: end of file
Fitxategia irekitzen...
Lehenengo lerroa: 8000
```

Hiru lerro irakurri baino lehen bukatu da fitxategia. hClose funtzioa ez da exekutatu.

7.15.3.3. hIsEOF: fitxategien bukaera kontrolatzeko

Orain lerro guztiak aurkeztu nahi dira, baina fitxategiaren bukaera kontrolatuz. Beraz, fitxategiaren bukaerara iritsitakoan gelditu egin beharko da errorerik sortu gabe. Fitxategiaren bukaera detektatzeko, System.IO moduluan hIsEOF funtzioa daukagu. Funtzio horren mota honako hau da:

```
hIsEOF :: Handle -> IO Bool
```

Fitxategi baterako sarbidea emanda, IO Bool motako erantzuna itzuliko du hIsEOF funtzioak. Bool motako zatia eskuratzeko <- eragilea erabili beharko da. Horrela lortutako balio Boolearrak fitxategiaren bukaeran al gauden adieraziko digu.

i3 funtzioa i funtzioaren orokorpen bat da eta lehenengo hiru lerroak aurkeztu beharrean lerro guztiak aurkeztuko ditu. Hori egiteko, i4 funtzio laguntzailea erabiliko du:

```
i3 :: FilePath -> IO()
i3 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      putStrLn ("Lerro bakoitzeko edukia honako hau da: \n")
      i4 sarbidea 1
      putStrLn "Fitxategia ixten..."
      hClose sarbidea

i4 :: Handle -> Integer -> IO()
i4 sarbide_izena lerro_zenbakia =
    do
      bukaera <- hIsEOF sarbide_izena
      if bukaera == True
        then putStrLn "Fitxategiaren bukaera."
        else do
               lerroa <- hGetLine sarbide_izena
               putStrLn ("Lerroa " ++ (show lerro_zenbakia) ++ ": \n" ++ lerroa)
               i4 sarbide_izena (lerro_zenbakia + 1)
```

i4 funtzioak fitxategi baterako sarbidea jasoko du lehenengo argumentu gisa eta sarbide horri dagokion fitxategiko lerro guztiak aurkeztuko banan-banan eta lerro-zenbakia adieraziz. Bigarren argumentuak lerro-zenbakia kontrolatzeko balio du.

Demagun fitxategia10.txt fitxategiaren edukia honako hau dela:

fitxategia10.txt
40 100 20 700 50

`i3 "fitxategia10.txt"` exekutatutakoan, pantailan honako hau ikusiko da:

Pantaila

```
Fitxategia irekitzen...
Lerro bakoitzeko edukia honako hau da:

Lerroa 1:
40
Lerroa 2:
100
Lerroa 3:
20
Lerroa 4:
700
Lerroa 5:
50
Fitxategiaren bukaera.
Fitxategia ixten...
```

Bestalde, fitxategia11.txt fitxategiaren balioa honako hau baldin bada:

fitxategia11.txt
8000

`i3 "fitxategia11.txt"` exekutatutakoan, pantailan honako hau erakutsiko da:

Pantaila

```
Fitxategia irekitzen...
Lerro bakoitzeko edukia honako hau da:

Lerroa 1:
8000
Fitxategiaren bukaera.
Fitxategia ixten...
```

7.15.3.4. hGetContents: fitxategi bat osorik irakurtzeko

Fitxategi baten eduki osoa dena batera irakurtzeko aukera ematen duen funtzio bat ere badugu System.IO moduluan. hGetContents funtzioa da eta fitxategiaren eduki guztia karaktere-kate gisa itzuliko du. Funtzio horren mota honako hau da:

```
hGetContents :: Handle -> IO String
```

Fitxategi baterako sarbidea emanda, hau da, Handle motako elementu bat emanda, IO String motako balio bat itzuliko du. IO kentzeko eta String motako balioa eskuratzeko <- eragilea erabili beharko da. String motako balioa sarbideari dagokion fitxategiaren eduki osoa izango da.

Orain definituko den i5 funtzioak sarrerako datu gisa fitxategi baten izena jasoko du eta hGetContents funtzioa erabiliko du fitxategi horren eduki osoa eskuratzeko. Bukatzeko, eduki hori pantailan aurkeztuko da.

```
i5 :: FilePath -> IO()
i5 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea
      putStrLn ("Edukiak:\n" ++ edukia)
      putStrLn "Fitxategia ixten..."
      hClose sarbidea
```

Irakurritako fitxategiaren eduki osoa 'edukia' izeneko aldagaian gordeko da karaktere-kate gisa.

fitxategia10.txt fitxategiaren balioa honako hau baldin bada:

fitxategia10.txt
40 100 20 700 50

`i5 "fitxategia10.txt"`, exekutatu ondoren pantailan honako hau erakutsiko da:

Pantaila

`Fitxategia irekitzen...`
`Edukiak:`
`40`
`100`
`20`
`700`
`50`
`Fitxategia ixten...`

hGetContents funtzioaren bidez fitxategiaren eduki osoa karaktere-kate batean lortu ondoren, lerroak bereiztea nahi izanez gero, Data.String moduluko 'lines' funtzioa erabil dezakegu. 'lines' funtzioa aurreko atalean aurkeztu da.

Oraintxe definituko den i6 funtzioak, fitxategi baten izena sarrerako datu gisa emanda, hGetContents funtzioa erabiliz fitxategi horren eduki osoa lortuko du 'lines' funtzioa erabiliz lerroak bereiziko ditu. Bukatzeko, lerroak banan-banan aurkeztuko ditu lehenago definitu den h6 funtzioa berriro erabiliz.

```
Import System.IO -- openFile, hGetContents eta hClose erabili ahal izateko
import Data.List – h6 funtzioan genericLength funtzio aurredefinitua erabiltzeko
import Data.String  -- i6 funtzioan lines funtzio aurredefinitua erabiltzeko

i6 :: FilePath -> IO()
i6 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea
      let lerroka = (lines edukia)
      putStrLn ("Lerro bakoitzeko edukia honako hau da: \n")
      h6 lerroka 1
      putStrLn "Fitxategia ixten..."
      hClose sarbidea

h6 :: [String] -> Integer -> IO()
h6 kate_zerrenda lerro_zenbakia =
    do
      if (genericLength kate_zerrenda) == 0
        then putStrLn "Hutsa"
        else if (genericLength kate_zerrenda) == 1
              then putStrLn ((show lerro_zenbakia) ++ ". lerroa" ++ ": \n" ++ (head kate_zerrenda))
              else do
                  putStrLn ((show lerro_zenbakia) ++ ". lerroa" ++ ": \n" ++ (head kate_zerrenda))
                  h6 (tail kate_zerrenda) (lerro_zenbakia + 1)
```

Demagun fitxategia10.txt fitxategiaren edukia honako hau dela:

fitxategia10.txt
40 100 20 700 50

`i6 "fitxategia10.txt"` exekutatu ondoren, pantailan honako hau erakutsiko da:

Pantaila
Fitxategia irekitzen... Lerro bakoitzeko edukia honako hau da: 1. lerroa: 40 2. lerroa: 100 3. lerroa: 20 4. lerroa: 700 5. lerroa: 50 Fitxategia ixten...

7.15.3.5. hGetChar: karaktere bat irakurtzeko

Fitxategi batetik datuak irakurtzeko, hGetLine eta hGetContents funtzioez gain, System.IO moduluan hGetChar funtzioa ere badugu. Funtzio honek karaktere bat irakurtzeko balio du. Funtzio horren mota honako hau da:

```
hGetChar :: Handle -> IO Char
```

Beraz, sarrerako datu gisa fitxategi baterako sarbidea hartuko du IO Char motako elementu bat itzuliko du. Elementu horretako Char motako osagaia, hau da, karakterea eskuratzeko <- eragilea erabili beharko da.

Orain j funtzio definituko dugu. Funtzio horrek, fitxategi baten izena emanda, fitxategiko lehenengo bi karaktereak eskuratuko ditu eta pantailan aurkeztuko ditu.

```
j :: FilePath ->IO()
j fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      karak <- hGetChar sarbidea
      putStr "Lehenengo karakterea: "
      putChar karak
      karak <- hGetChar sarbidea
      putStr "\nBigarren karakterea: "
      putChar karak
      putStrLn "\nFitxategia ixten..."
      hClose sarbidea
```

Demagun fitxategia12.txt fitxategiaren edukia honako hau dela:

fitxategia12.txt
aei ou

`j "fitxategia12.txt"` exekutatutakoan, pantailan honako hau agertuko da:

Pantaila

`Fitxategia irekitzen...`
`Lehenengo karakterea: a`
`Bigarren karakterea: e`
`Fitxategia ixten...`

Baina j funtzioari argumentu gisa emandako fitxategiak bi karaktere baino gutxiago baldin baditu, exekuzio-errorea gertatuko da.

Demagun fitxategia13.txt fitxategiaren edukia honako hau dela:

fitxategia13.txt
a

`j "fitxategia13.txt"` exekutatutakoan, pantailan honako hau erakutsiko da:

Pantaila

`*** Exception: fitxategia13.txt: hGetChar: end of file`
`Fitxategia irekitzen...`
`Lehenengo karakterea: a`

Fitxategiaren bukaera ez denez kontrolatu eta bigarren karaktererik ez dagoenez fitxategia13.txt fitxategian, exekuzio-errorea sortu da. Fitxategia ixteko balio duen hClose funtzioa ez da exekutatu.

Orain j funtzioaren orokorpena den j2 funtzioa definituko dugu. j2 funtzioak, fitxategi baten izena emanda, fitxategi horretako karaktere guztiak banan-banan irakurriz eta pantailan aurkeztuz joango da. Laguntzaile gisa j3 funtzioa erabiliko da.

```
j2 :: FilePath -> IO()
j2 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      putStr ("Karakterez karaktere edukia honako hau da: ")
      j3 sarbidea 1
      putStrLn "\nFitxategia ixten..."
      hClose sarbidea

j3 :: Handle -> Integer -> IO()
```

```
j3 sarbide_izena lerro_zenbakia =
    do
      bukaera <- hIsEOF sarbide_izena
      if bukaera == True
       then putStr "\nFitxategiaren bukaera."
       else do
              karak <- hGetChar sarbide_izena
              putStr ("\n" ++ (show lerro_zenbakia) ++ ". karakterea: ")
              putChar karak
              j3 sarbide_izena (lerro_zenbakia + 1)
```

`j2 "fitxategia12.txt"` exekutatutakoan, pantailan honako hau agertuko da:

Pantaila

```
Fitxategia irekitzen...
Karakterez karaktere edukia honako hau da:
1. karakterea: a
2. karakterea: e
3. karakterea: i
4. karakterea:

5. karakterea: o
6. karakterea: u
Fitxategiaren bukaera.
Fitxategia ixten...
```

Hor ikus dezakegu lerro bukaera adierazten duen '\n' karakterea ere beste karakterak bezala tratatu eta zenbatu dela.

j4 eta j5 funtzioak j2 eta j3 funtzioen antzekoak dira. Baina j5 funtzioak lerro bukaerako '\n' karakterea ez du beste karaktereak bezala tratatzen. Izan ere, ez du aurkeztuko eta ez du zenbatuko.

```
j4 :: FilePath -> IO()
j4 fitx =
    do
      putStrLn "Fitxategia irekitzen..."
      sarbidea <- openFile fitx ReadMode
      putStr ("Karakterez karaktere edukia honako hau da: ")
      j5 sarbidea 1
      putStrLn "\nFitxategia ixten..."
      hClose sarbidea

j5 :: Handle -> Integer -> IO()
j5 sarbide_izena lerro_zenbakia =
    do
      bukaera <- hIsEOF sarbide_izena
      if bukaera == True
       then putStr "\nFitxategiaren bukaera."
```

```
    else do
        karak <- hGetChar sarbide_izena
        if karak == '\n'
         then do
            putStr ""  -- Testu hutsa
            j5 sarbide_izena lerro_zenbakia
         else do
            putStr ("\n" ++ (show lerro_zenbakia) ++ ". karakterea: ")
            putChar karak
            j5 sarbide_izena (lerro_zenbakia + 1)
```

`j4 "fitxategia12.txt"` exekutatutakoan, pantailan honako hau erakutsiko da:

Pantaila

```
Fitxategia irekitzen...
Karakterez karaktere edukia honako hau da:
1. karakterea: a
2. karakterea: e
3. karakterea: i
4. karakterea: o
5. karakterea: u
Fitxategiaren bukaera.
Fitxategia ixten...
```

Hor ikus daiteke lerro bukaerako karakterea ez dela beste karaktere bat balitz bezala aurkeztu. Karaktere hori ez da aurkeztu eta ez da zenbatu.

7.15.3.6. Fitxategietan idazteko: hPutChar, hPutStr eta hPutStrLn

Fitxategi batetik datuak irakurtzeko System.IO moduluan ditugun oinarrizko aukerak aurkeztu ondoren, fitxategietan idazteko ditugun oinarrizko funtzioak aurkeztuko dira orain. Honako funtzio hauek azalduko ditugu orain: hPutChar, hPutStr eta hPutStrLn. hPutChar funtzioak karaktere bat idatziko du. hPutStr funtzioak karaktere-kate bat idatziko du. Azkenik, hPutStrLn funtzioak karaktere-kate bat idatziko du eta lerro berrira jauzia egingo du, hau da, lerroz aldatuko da. Funtzio hauek erabiltzeko fitxategi bat irekitzen denean, WriteMode edo AppendMode aukera hautatu beharko da. WriteMode hautatzen bada, fitxategian aurretik zegoen guztia galdu egingo da. AppendMode aukeratzen bada, fitxategian aurretik zegoena mantendu egingo da eta eduki berria fitxategiaren bukaeran erantsiko da.

Aipatutako hiru funtzio horien motak honako hauek dira:

```
hPutChar :: Handle -> Char -> IO ()

hPutStr :: Handle -> String -> IO ()

hPutStrLn :: Handle -> String -> IO ()
```

Orain k izeneko funtzioa definituko dugu. Funtzio horrek, argumentu gisa emandako fitxategian 'W', 'X', 'Y' eta 'Z' karaktereak idatziko ditu. Hasteko, fitxategiak aurretik zuen edukia aurkeztu nahi denez, irakurtzeko eran ireki beharko da fitxategia. Fitxategia horrela ireki ondoren, edukia irakurri, edukia pantailan aurkeztu eta fitxategia itxi egin beharko da. Gero, irakurtzeko eran ireki beharko da fitxategia eta aipatutako lau karaktereak idatziko dira. k funtzioak lau karaktere horiek banan-banan idatziko ditu hPutChar erabiliz. Beste aukera bat hPutStr erabiliz "WXYZ" karaktere-katea idaztea izango litzateke. Karaktereak idatzi ondoren fitxategia itxi egin beharko da. Bukatzeko, fitxategiak bukaeran duen edukia pantailan aurkeztukoda. Horretarako, irakurtzeko eran ireki beharko da fitxategia berriro ere, gero fitxategiaren edukia irakurri eta eduki hori pantailan aurkeztu beharko da eta fitxategia itxi egin beharko da.

```
k :: FilePath -> IO()
k fitx =
    do
      putStrLn "Fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren hasierako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten..."
      hClose sarbidea1
--
      putStrLn "Fitxategia idazketarako irekitzen..."
      sarbidea2 <- openFile fitx WriteMode
      hPutChar sarbidea2 'W'
      hPutChar sarbidea2 'X'
      hPutChar sarbidea2 'Y'
      hPutChar sarbidea2 'Z'
      putStrLn "Fitxategia idazketarako ixten..."
      hClose sarbidea2
--
      putStrLn "Fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren bukaerako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten..."
      hClose sarbidea1
```

k funtzioan, irakurterako sarbidea eta idazketarako sarbidea bereizteko bi sarbide erabili dira: 'sarbidea1' eta 'sarbidea2'.

Demagun fitxategia14.txt fitxategiaren edukia honako hau dela:

fitxategia14.txt
aei ou

k "fitxategia14.txt" exekutatutakoan, pantailan honako hau ikusiko da:

```
Fitxategia irakurketarako irekitzen...
Fitxategiaren hasierako edukia honako hau da:
aei
ou
Fitxategia irakurketarako ixten...
Fitxategia idazketarako irekitzen...
Fitxategia idazketarako ixten...
Fitxategia irakurketarako irekitzen...
Fitxategiaren bukaerako edukia honako hau da:
WXYZ
Fitxategia irakurketarako ixten...
```

k2 funtzioa k funtzioaren antzekoa da baina hPutChar erabili beharrean hPutStr funtzioa erabiltzen du eta, gainera, "uvw" eransten du fitxategiaren edukiaren bukaeran. Horretarako, datuak eransteko eran irekiko da fitxategia eta ez datuak idazteko eran.

```
k2 :: FilePath ->IO()
k2 fitx =
    do
      putStrLn "Fitxategia irakurketarako irekitzen... "
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren hasierako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten... "
      hClose sarbidea1
--
      putStrLn "Fitxategia eransketarako irekitzen... "
      sarbidea2 <- openFile fitx AppendMode
      hPutStr sarbidea2 "\nuvw"
      putStrLn "Fitxategia eransketarako ixten... "
      hClose sarbidea2
--
      putStrLn "Fitxategia irakurketarako irekitzen... "
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren bukaerako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten... "
      hClose sarbidea1
```

"uvw" katea lerro berri batean ipintzeko, "\nuvw" erantsi zaio fitxategiaren edukiari. Hor, '\n' karaktereak lerro-jauzia eragingo du.

fitxategia15.txt fitxategiaren edukia honako hau baldin bada:

fitxategia15.txt
aei ou

k2 "fitxategia15.txt" exekutatutakoan, pantailan honako hau ikusiko da:

```
Fitxategia irakurketarako irekitzen...
Fitxategiaren hasierako edukia honako hau da:
aei
ou
Fitxategia irakurketarako ixten...
Fitxategia eransketarako irekitzen...
Fitxategia eransketarako ixten...
Fitxategia irakurketarako irekitzen...
Fitxategiaren bukaerako edukia honako hau da:
aei
ou
uvw
Fitxategia irakurketarako ixten...
```

k3 funtzioa k2 funtzioaren antzekoa da, baina sarrerako datutzat fitxategi baten izena eta testu bat jasoko ditu. k3 funtzioak testu hori erantsi beharko dio fitxategiaren edukiari.

```
k3 :: FilePath -> String -> IO()
k3 fitx testua =
    do
      putStrLn "Fitxategia irakurketarako irekitzen... "
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren hasierako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten... "
      hClose sarbidea1
--
      putStrLn "Fitxategia eransketarako irekitzen... "
      sarbidea2 <- openFile fitx AppendMode
      hPutStr sarbidea2 ("\n" ++ testua)
      putStrLn "Fitxategia eransketarako ixten... "
      hClose sarbidea2
--
      putStrLn "Fitxategia irakurketarako irekitzen... "
      sarbidea1 <- openFile fitx ReadMode
      edukia <- hGetContents sarbidea1
      putStrLn ("Fitxategiaren bukaerako edukia honako hau da: ")
      putStrLn edukia
      putStrLn "Fitxategia irakurketarako ixten... "
```

```
    hClose sarbidea1
```

'\n' karakterea erabili behar da testua lerro berrian eransteko.

fitxategia16.txt fitxategiaren edukia honako hau baldin bada:

fitxategia16.txt
```
aei
ou
```

k3 "fitxategia16.txt" "vvv" exekutatutakoan, pantailan honako hau erakutsiko da:

```
Fitxategia irakurketarako irekitzen...
Fitxategiaren hasierako edukia honako hau da:
aei
ou
Fitxategia irakurketarako ixten...
Fitxategia eransketarako irekitzen...
Fitxategia eransketarako ixten...
Fitxategia irakurketarako irekitzen...
Fitxategiaren bukaerako edukia honako hau da:
aei
ou
vvv
Fitxategia irakurketarako ixten...
```

Orain 'fbikoitza' izeneko funtzioaren definizioa emango da. Funtzio horri bi fitxategiren izenak emqango zaizkio sarrerako datu gisa. Lehenengo fitxategian lerro bakoitzeko zenbaki oso bat egongo dela suposatuz, balio horiek bi zenbakiaz biderkatuta bigarren fitxategian gordeko ditu. Laguntzaile gisa 'fbikoitza_lag' funtzioa definituko da.

```
fbikotza :: FilePath -> FilePath -> IO()
fbikoitza fitx1 fitx2 =
    do
      putStrLn "Lehenengo fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx1 ReadMode
      putStrLn ("Lehenengo fitxategiaren edukia honako hau da: \n")
      edukia1 <- hGetContents sarbidea1
      putStrLn edukia1
      putStrLn "Lehenengo fitxategia irakurketarako ixten..."
      hClose sarbidea1
--
      putStrLn "Lehenengo fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx1 ReadMode
      putStrLn "Bigarren fitxategia idazketarako irekitzen..."
      sarbidea2 <- openFile fitx2 WriteMode
      fbikoitza_lag sarbidea1 sarbidea2
      putStrLn "Lehenengo fitxategia irakurketarako ixten..."
      hClose sarbidea1
      putStrLn "Bigarren fitxategia idazketarako ixten..."
```

```
    hClose sarbidea2
--
    putStrLn "Bigarren fitxategia irakurketarako irekitzen..."
    sarbidea2 <- openFile fitx2 ReadMode
    putStrLn ("Bigarren fitxategiaren bukaerako edukia honako hau da: \n")
    edukia2 <- hGetContents sarbidea2
    putStrLn cedukia2
    putStrLn "Bigarren fitxategia irakurketarako ixten..."
    hClose sarbidea2


fbikoita_lag :: Handle -> Handle -> IO()
fbikoitza_lag sarbide_izena1 sarbide_izena2 =
  do
    bukaera <- hIsEOF sarbide_izena1
    if bukaera == True
      then putStrLn "Fitxategiaren bukaera."
      else do
            lerroa <- hGetLine sarbide_izena1
            let zenb = (read lerroa :: Integer)
            hPutStrLn sarbide_izena2 (show (zenb * 2))
            fbikoitza_lag sarbide_izena1 sarbide_izena2
```

fbikoitza funtzioak hasieran lehenengo fitxategiaren eduki osoa irakurriko du. Ondorioz, fitxategi horretarako sarbidea fitxategiaren bukaeran kokatuta gelditzuko da. Lehenengo fitxategia berriro irakurtzeko, hau da, lerroz lerro irakurtzeko, fitxategia itxi eta berriro ireki beharko da.

Demagun salmentak.txt fitxategiaren edukia honako hau dela:

salmentak.txt
40 100 20 700 50

`fbikoitza "salmentak.txt" "bikoitza.txt"` exekutatutakoan, bikoitza.txt fitxategiaren edukia honako hau izango da:

bikoitza.txt
80 200 40 1400 100

Eta pantailan honako hau erakutsiko da:

Pantaila

```
Lehenengo fitxategia irakurketarako irekitzen...
Lehenengo fitxategiaren edukia honako hau da:

40
100
20
700
50
Lehenengo fitxategia irakurketarako ixten...
Lehenengo fitxategia irakurketarako irekitzen...
Bigarren fitxategia idazketarako irekitzen...
Fitxategiaren bukaera.
Lehenengo fitxategia irakurketarako ixten...
Bigarren fitxategia idazketarako ixten...
Bigarren fitxategia irakurketarako irekitzen...
Bigarren fitxategiaren bukaerako edukia honako hau da:

80
200
40
1400
100

Bigarren fitxategia irakurketarako ixten...
```

Para terminar este apartado sobre funciones disponibles en System.IO moduluan fitxategiekin aritzeko ditugun oinarrizko funtzioei buruzko atala bukatzeko, 'fbatura' izeneko funtzioa definituko dugu. 'fbatura' funtzioak bi fitxategiren izenak jasoko ditu sarrerako datu gisa. Lehenengo fitxategiak lerro bakoitzean zenbaki oso bat izango duela suposatuz, zenbaki horien batura kalkulatuko da eta emaitza bigarren fitxategian gordeko da. 'fbatura' funtzioak 'fbatura_lag' funtzioa erabiliko du laguntzaile gisa.

```
fbatura :: FilePath -> FilePath -> IO()
fbatura fitx1 fitx2 =
    do
      putStrLn "Lehenengo fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx1 ReadMode
      putStrLn ("Lehenengo fitxategiaren edukia honako hau da: \n")
      edukia1 <- hGetContents sarbidea1
      putStrLn edukia1
      putStrLn "Lehenengo fitxategia irakurketarako ixten..."
      hClose sarbidea1
--
      putStrLn "Lehenengo fitxategia irakurketarako irekitzen..."
      sarbidea1 <- openFile fitx1 ReadMode
      putStrLn "Bigarren fitxategia idazketarako irekitzen..."
      sarbidea2 <- openFile fitx2 WriteMode
      b <- (fbatura_lag sarbidea1 0)
      hPutStr sarbidea2 (show b)
      putStrLn "Lehenengo fitxategia irakurketarako ixten..."
```

```
      hClose sarbidea1
      putStrLn "Bigarren fitxategia idazketarako ixten..."
      hClose sarbidea2
--
      putStrLn "Bigarren fitxategia irakurketarako irekitzen..."
      sarbidea2 <- openFile fitx2 ReadMode
      putStrLn ("Bigarren fitxategiaren bukaerako edukia honako hau da: \n")
      edukia2 <- hGetContents sarbidea2
      putStrLn edukia2
      putStrLn "Bigarren fitxategia irakurketarako ixten..."
      hClose sarbidea2


fbatura_lag :: Handle -> Integer -> IO Integer
fbatura_lag sarbide_izena batura =
    do
      bukaera <- hIsEOF sarbide_izena
      if bukaera == True
        then return batura
        else do
               lerroa <- hGetLine sarbide_izena
               let zenb = (read lerroa :: Integer)
               fbatura_lag sarbide_izena (batura + zenb)
```

salmentak.txt fitxategiaren edukia honako hau baldin bada:

salmentak.txt
40 100 20 700 50

`fbatura "salmentak.txt" "batura.txt"` exekutatutakoan, batura.txt fitxategiaren edukia honako hau izango da:

batura.txt
910

Eta pantailan honako hau aurkeztuko da:

Pantaila

`Lehenengo fitxategia irakurketarako irekitzen...`
`Lehenengo fitxategiaren edukia honako hau da:`
 	
`40`
`100`
`20`
`700`
`50`

```
Lehenengo fitxategia irakurketarako ixten...
Lehenengo fitxategia irakurketarako irekitzen...
Bigarren fitxategia idazketarako irekitzen...
Lehenengo fitxategia irakurketarako ixten...
Bigarren fitxategia idazketarako ixten...
Bigarren fitxategia irakurketarako irekitzen...
Bigarren fitxategiaren bukaerako edukia honako hau da:

910
Bigarren fitxategia irakurketarako ixten...
```