PROGRAMEN EGIAZTAPENA ETA ERATORPENA

Xabier Arregi
Arantza Díaz de Ilarraza
Paqui Lucio

UDAKO EUSKAL UNIBERTSITATEA
Bilbo, 1993

ISBN: 84-86967-50-3
Lege-gordailua: BI-22-92

Inprimategia: BOAN, S.A. Padre Larramendi 2 BILBO

Azala: Julio Pardo

Banatzaileak: UEU. General Concha 25, 6. BILBO
Zabaltzen: Igarabide, 88 DONOSTIA

AURKIBIDEA

SARRERA

Programazioaren hastapenetan makina-lengoaian idazten ziren programak. Ez zegoen beste aukerarik, konputagailuak ezin bait zezakeen kode bitarrean idatzikoaz besterik ulertu. Sasoi hartan nola-halako notazio sinbolikoa erabiltzen zen makina eragiketak eta, hauek kateatuz, programak idazteko.

Pixkanaka, mihiztadura-lengoaiak sortu ahala, kode sinboliko hori makina-kodetik urruntzen hasi zen. Lengoaia hauetan idatzitako kodea programa itzultzaileen bidez (mihiztatzaileak) makina-kodera pasatzen zen. Makinatik urrundu bai, zertxobait urruntzen zen mihiztadura-kodea, baina hala ere menpekotasun edo lotura estua zeukan makinarekiko; hain estua ezen lengoaiako primitiboak makinaren oinarrizko hardware-eragiketak bait ziren. Hitz gutxitan esateko, programagintza astuna zen, metodorik gabea, errore-arrisku oso altukoa eta hauen detekzio eta arazketa benetan nekosoa.

Gauzak honela, alegiazko makina erosoago baten premia nabarmendu zen, edo, nahi bada, erabilgarriago gertatuko zen programazio-lengoaia batena.

Diseinu-ahaleginak 1945ean hasi ziren, eta lan horien fruitu gisara 50. hamarkadaren bukaera aldera agertu ziren estraineko programazio-lengoaiak. FORTRAN izan zen lehenengoa, 1954 eta 1957.aren artean John Backus eta bere IBM taldeak garatua, eta benetan eragin handikoa. Harrera eszeptikoa egin zitzaion, ez bait zen uste hain goi--mailako lengoaia eraginkorra izan zitekeenik, eta horregatixe ez zen inguru akademikoetatik atera harik eta Backus-en taldeak konpiladore egokia sortu eta behe--mailako kodeekin bezain emaitza onak lortu zituen arte. Orduantxe eman zuten ameto eszeptikoek. Egia esatera, FORTRAN horrek mihiztadura-lengoaien ezaugarri askotxo zituen.

Garaitsu hartan (50.en bukaera eta 60.en hasieran) lengoaia ugari jaio zen: ALGOL, COBOL, APL, BASIC, LISP. Guztiek, LISP-ek izan ezik, bazuten FORTRAN lengoaiaren eraginik eta, modu batera edo bestera, hura hobetzeko asmotan sortu ziren: programen irakurgarritasuna zela, datuen erabilpena zela, e. a.

Ugalketa horrek eztabaidari eta, bidenabar, lengoaien arteko alderaketari eman zion bidea. Ingurune honetan sortu zen programazio-lengoaia unibertsalaren ideia eta halaxe jaio zen 1965ean PLI lengoaia. Esperientzia honek ederki erakutsi zuen benetan zaila zela "denetarako" balio zuen lengoaia ikasi eta ondo erabiltzea. Nabaria zen aplikazio-arlo desberdinetan lengoaia berezituak behar zirela.

Garai hartako literatura ("Programming in Basic", "Programming in Fortran IV") ikusi besterik ez dago konturatzeko benetan garrantzizkoak lengoaiak zirela. Programatzen jakitea programazio-lengoaiak jakitea zen, eta ez besterik. Programatzaileak bere kaxa ikasten zituen oinarrizko arauak, askotan aritzeak ematen duen eskarmentura mugatzen zirenak.

60. hamarkadan aplikazioen konplexutasuna, tamaina eta garrantziak gora egin zuten arren, ez zen gauza bera gertatu programatzaileen iaiotasun eta tresneriarekin. Programa gutxi eta eskasak egiten ziren, eta denbora gehiago behar izaten zen errore-arazketan programak sortzen baino. Egoera penagarri horrek bultzatuta, 1968ko Urrian "Software Engineering" izeneko konferentzia ospatu zen Garmisch-en (Alemanian) NATOren babespean. Bilera hartan "Softwarearen krisiaz" hitz egin zen, hau da, beharren eta metodoen arteko desorekak sortutako krisiaz. Bildutakoak, akademiko nahiz industri gizonak, adostasun batera ailegatu ziren: "Softwaregintza krisian badago programazioa orain artean gaizki ulertua izan delako da". Planteamendu horrek aurrekoarekin apurtzea eta bide berrietatik abiatzea ekarri zuen. Urtebete barru, 1969an, IFIP elkarteak Working Group 2.3 lan-taldea sortzea erabaki zuen, "programazio-metodologia" lantzera dedikatuko zena, hain zuzen ere.

Talde horrek erro-errotiko aldaketak proposatu zituen: utikan artisau lana, egin dezagun programazio zientifikoagoa! Ikusmolde horrek bi oinarri ditu:

1.- Programa baten ezaugarri nagusiak honako hauek izan behar dute: zuzentasuna, argitasuna, aldagarritasuna eta eraginkortasuna.

2.- Programatzaileak behar-beharrezkoa du bere lana erraztuko duen tresneria eta oinarri teorikoa.

Pentsamolde honen haritik eratorritako programazio-estiloari "Programazio Egituratu" deitu zitzaion. Bazirudien, hasieran izandako zenbait eztabaidaren arabera behinipehin, programazio egituratua *go to* agindurik gabe programatzea baizik ez zela. Luzagabe argitu zen, ordea, hori baino zerbait gehixeago zela programazio egituratuaz ulertu behar zena.

Hortik aurrera ez ziren falta izan era guztietako ekarpenak, hala nola, programazio--lengoaiak, programen disenurako metodologiak, e. a.

1970ean PASCAL programazio-lengoaia jaio zen, N. Wirth-ek diseinatua. Lengoaia honetan notazio hobeak, kontrol-egitura argiagoak eta datuen definizio eta maneiurako erraztasunak bildu nahi ziren. Gerora, PASCAL lengoaia hainbat programazio-lengoaia agintzaileren aurrekaria gertatu da, esate baterako, MODULA-2 eta ADArena.

PASCAL-arekin batera abstrakzioaren kontzeptua programazioaren mundura ailegatu zen. Abstrakzio funtzionalak ebatzi beharreko problemaren konplexutasuna

zatikatzeko aukera ematen zuen. Problemari bere osotasunean aurre egin beharrean, zatika, aldiro problemaren aspektu batzuk ebaztean, eta gainerakoei itzuri egitean, datza abstrakzioa. Honelaxe plazaratu zuen N. Wirth-ek ondoz-ondoko finketen metodoa 1971n. Ordutik aurrera programazio-metodologiaren funtsa hauxe izan da: programatzea ez da soilik programak idaztea. Programen idazketa, izatekotan, analisi eta diseinu prozesuaren azken urratsa da, eta prozesu hori metodo ordenatu batez burutu behar da, ez nola edo hala. Ondoz-ondoko finketen metodoak berebiziko eragina izan du geroztik formulatu diren metodoetan, esate baterako, gestiorako programen diseinuan ere beheranzko metodoa erabiltzen da, Warnier metodoan kasu.

Programen zuzentasuna egiaztatzeko metodoek ere, ez soilik probetan oinarritutakoek, piztualdi polita ezagutu zuten testuinguru honetan. Lehendik ere, 1949an, A. Turing-ek plazaratua zuen behar hori, bereziki bere metodoa aurkeztu zuenean. Bada bai ildo honetan lan egin duen aitzindaririk, besteak beste, McCarthy, Naur eta Floyd aipa daitezke. Floyd-ek, 1967an, Turing-en ideian oinarritutako metodoa aurkeztu zuen, baina asertzio gisa aldagaien balioak erlazionatzen zituzten formulak erabili zituen. Guzti horren buruan inor gutxik jartzen zuen zalantzan programen fidagarritasuna hobetzearen beharra eta, halaber, testak soil geratu eta ez zirela nahikoa. Dijkstra-k zioen bezala, probek programak erroreak badituela froga dezakete, baina ez ordea erroregabea denik. Horretan dago, hain zuzen, koxka; horregatik da hain interesgarria programak egiaztatzea.

1969an, Hoare-k, Floyd-en ideiak berreskuratuz, axioma eta inferentzi erregelatako sistema formala eratu zuen. Sistema honek programen zuzentasun formala frogatzeko balio du eta asertzio inbarianteen metodoaren euskarria da.

Geroztik programen zuzentasuna formalki frogatzeko zenbait metodo azaldu da, hala nola, "aldizkako asertzioena" edo semantika denotazionalean oinarritutakoa. Baina guztietan erabiliena Hoare-ren kalkulua izan da ezbairik gabe.

Handik laisterrera, ondoz-ondoko birfinketen diseinu-metodoa eta asertzio inbarianteen egiaztapen-metodoa ezkontzeko asmoa sortu zen. Asmoa sinplea zen: arrazoizkoa da, eta eraginkorragoa gainera, programa egin ahala zuzena dena frogatzea, bi eginkizunak, diseinua eta egiaztapena, ideia beretsuen garapenak bait dira. Horrez gain, programagintza-metodo horrek aukera aparta eskaintzen du diseinuaren propietateak eta erabilitako ideiak ondo dokumentatuta uzteko, eta hori bai dela eskertzekoa programen mantenimendu-fasean.

Dijkstra-k programen eratorpen formala izeneko metodoa planteatu zuen 1975ean, hau ere Hoare-ren kalkulutik eratorritakoa. Metodo horren arabera, programaren helburua espezifikazio jakin bat betetzea da eta horretara zuzendu behar da diseinua. Espezifikazioaren ondoko baldintzak, batetik, lortu nahi den emaitza nolakoa den adieraziko digu eta aurreko baldintzak, bestetik, datuek bete behar dituzten murriztapenak

finkatuko ditu. Programazioa hala planteatuz gero ondoko baldintza bihurtzen da diseinuaren gidari, baina, hori bai, diseinuak eskatzen dituen gutxienezko betebeharrak aurreko baldintzak biltzen dituela aurreikusiz. Funtsezko ideia eta formalismoa Dijkstra-k azaldu bazuen ere, Gries izan zen espezifikaziotik abiatuta programak eratortzeko metodoa gehixeago landu eta ezagutzera eman zuena.

Programen eraldakuntzak ere badu zeresanik aipatzen ari garen arrazonamendu eta zuzentasunaren egiaztapenean oinarritutako metodologian. Normalean, eraldatu, programa sinpleak eraldatzen dira, espezifikazioa betetzen dutenak baina eraginkorrak ez direnak. Programa horien semantika babestuz (zuzentasunari eutsiz) baliokide eraginkorragoak lortu nahi izaten dira. Ez da, beraz, beheranzko metodoa, horizontala baizik. Gehienetan, programa eraginkorrak ulergaitzak dira, eta zaila izaten da zuzentasunari buruz arrazonatzea. Horrelakoetan, askozaz erosoagoa gertatzen da jatorrizko programa zuzena zela eta eraldaketak zuzentasunari eusten diola egiaztatzea.

Egia da programa-eraldakuntzak programa batetik besterako pausoa ematen duela eta ez espezifikaziotik programarakoa. Eraldakuntzan ere badira, ordea, espezifikazioaren eta egiaztapenaren beharra duten problema batzuk, txukun ebatziko badira behintzat. Eraldaketak programen zuzentasuna kontserbatzen duela frogatu nahi bada, esate batera, beharrezkoa da programaren espezifikazioa ezagutzea eta egiaztapen-teknikak erabiltzea. Gainera, zenbait eraldaketak ez du, murriztapenik ezarri ezean, zuzentasuna beti kontserbatzen.

Programa errekurtsiboak gehientsuenetan sinpleak eta ulerterrazak dira, baina baita ez-eraginkorrak ere. Ez da harritzekoa, beraz, errekurtsibo-iteratibo ereduko eraldakuntzak izatea gaur egun erabilienak eta landuenak.

Programen sintesia edo programazio automatikoa ere espezifikaziotik hasi eta programa sortzera jotzen duen metodoa da, baina kasu honetan adimen artifizialaren inguruko teknikak erabiltzen dira. Batzuetan, espezifikazioa lengoaia funtzionalean idazten denean, sintesia ez da programa-eraldakuntza mekanizatua baizik. Beste batzuetan, espezifikaziotik programa bat lortzen da zuzenean eta gero programa hori hobetu egiten da programa-eraldakuntzaz.

Programen egiaztapen-prozesuak dituen aspektu mekanikoenak automatizatzeko joerak, ikerkuntz adar berri bati eman zion sorrera egiaztapenaren alorrean. Programazio-inguruneetan tresna berri bat integratu nahi zen: egiaztapen-sistemak; eta hartara programen zuzentasun semantikoa egiaztatzeko erraztasunak eskaini. Hamaika ahalegin egin da 70.etik aurrera egiaztapen-metodo mekanizagarriak deskribatzeko eta egiaztapen-sistema batzuk eraiki ere egin dira, baina elkarrekintzazkoak eta esperimentalak dira oraingoz. Etorkizunean egiaztapen-sistemak programazio-ingurune integratuetako osagaiak izango dira, editoreak, konpiladoreak eta liburutegiak bezalaxe.

Abstrakzioarekin zerikusia duen beste programazio-metodo inportantea programa-eskematan oinarritzen dena da. Programa-eskema algoritmo generikoa da, ekintza, funtzio eta objektu abstraktuak (interpretatu gabeak) dauzkana.

Programa-eskemek ideia zahar batean dute jatorria: ebatzi nahi dugun problemak beste batekin erlazio estua badu, eta azken honen ebazpena ezaguna bada, ebatzi dezagun jatorrizkoa bigarrenean oinarrituta.

Azken finean, algoritmo askok portaera beretsua dutela aipatzen duten liburuak programa-eskemen erabileraz ari dira, nahiz eta ez duten diseinu-metodo hau esplizituki aipatzen.

Dijkstra-k "programa-familiak" aipatzen ditu egoki baino egokiago, hau da, algoritmo abstraktu batetik (oraindik birfintzeke dagoena) erator daitezkeen programak jotzen ditu senide eta hauek, bere ustetan, problema-mota (edo familia) bati aplika dakizkioke. Eskema bera, jatorrizko algoritmo abstraktua alegia, dokumentazio ezin aproposagoa da mantenimendu-faserako.

Eskemen bidezko programa-diseinurako problemak klaseka bildu behar dira, ebazpidean erabiltzen diren ezaugarri komunen arabera. Honela, behin problema klaseren batean sailkatuta, programa-eskema zehatz bat aukeratzen da ebazpenerako, eta eskema horretatik ekintzak eta objektu abstraktuak birfindu beste lanik gabe, programa lortzen da.

Orain artean azaldutako programazio-metodologiaren aspektu horietan oinarritu gara liburu honen edukia mamitzean. Esan behar da, gainera, programazioaren inguruko ikerkuntz lerro nagusiak ingurune horretan ari direla garatzen gaur egun.

Hemen jasotzen den materiala programazioko 2. mailara zuzenduta dago eta bertan programazio agintzailearen oinarrizko ezagutza duen irakurleari programazio metodikoaren bidea erakutsi nahi zaio. Garrantzizkoa deritzogu bide hau urratzeari, izan ere, Dijkstra-k esan zuen bezala, zaila bait da artisau-programazioan ohituta dagoenari programazio metodikoaren komenientzia ikustaraztea.

Gure iritziz, programatzea ez da "funtzionatzen" duten programak idaztea soilik. Hori baino areago, programei argitasuna, moldagarritasuna, dokumentazio aproposa, eraginkortasuna eta estilo horretako ezaugarriak eskatu behar zaizkie.

Programatzea ez da iharduketa nabaria, ezta sinplea ere. Ezin da edozein moduz programatu, baizik eta lagungarri gertatuko diren tresneria eta ezagutza egokiez baliatuta. Programen diseinu-prozesuak ideien antolaketa eta zorroztasunez egindako azterketa eskatzen du derrigor. Hala egiteak, zenbaitetan bederen, programazioa zaildu egiten duela eman lezake, baina egiaz diseinu-erroreak egiteko arriskua txikitu egiten du (hauek dira gainera detektatzeko eta zuzentzeko gaitzenak) eta ondorioz programazio-lana erraztu.

Horrez gain, txukun arrazonatu eta eratutako diseinua da izan daitekeen dokumentaziorik egokiena, eta aspektu inportanterik bada programazioan, horietako bat dokumentazioa da.

Behin problemaren ezaugarriak aztertuta, komeni da programazio-metodo egokiena aukeratzeko gai izatea. Baina horretarako beharrezkoa da metodo eta kontzeptu desberdinak noiz eta nola erabili behar diren jakitea.

Sarrera gisako honetan ukitu ditugun aspektu hauetako batzuk lantzen dira liburuan zehar, beti ere programa zuzen, argi, eta ulergarriak idatzi nahi dituenari laguntza eskaini nahian.

1. ESPEZIFIKAZIOA

1.1. ZER DA ESPEZIFIKAZIOA

Espezifikazioa, hiztegiek diotenez, egin beharreko lanaren deskribapen zehatza da. Baina, baieztapen hau emanez gero:

"... LA (lerro-amaiera) karakterearen bidez banantzen diren lerroez osatutako testua emanda, bertatik irazkin guztiak, hau da, `(´ eta `)´ karaktereen arteko sekuentzia guztiak ezabatzen dituen programa..."

Deskribapen zehatza al da? Bistan da deskribapena dela, ez zehatza ordea. Inork programa hau egin behar balu, edo egindakoren bat ulertu, berehala hasiko litzateke galdezka:

- Nola detektatzen da testu-amaiera?
- Egon al litezke irazkin okerrak? hau da, ixten ez diren parentesi irekiak edo alderantzizkoak?
- Litekeena al da irazkin kabiatuak egotea?, eta irazkin hutsak?
- Zer egiten du (edo zer egin beharko luke) programak balizko irazkin okerren aurrean?
- Irazkinak ezabatzean hutsunerik utzi behar al da? Ezezkoan, gehienez ere zenbat karaktere jar daitezke lerro berean?

Aldiz, aurreko enuntziatuaren ordez beste hau idatzi izan balitz:

datuak: Karakterez osatutako testua, non karaktereak honela sailkatzen bait dira:
- Testu-amaiera: TA
- Lerro-amaiera: LA
- `(´ eta `)´ parentesiak
- Karaktere arruntak: gainontzekoak, ondoko baldintza hauen pean:
  - TA behin agertzen da eta beti azkena
  - Lerro bakoitza, gehienez ere, 80 karakterez osatzen da.
  - `(´ eta `)´ karaktereen arteko sekuentziak kabia daitezke.

emaitzak: testua, non:
- Karaktere-sekuentzia sarrerako testuaren berdina bait da; `(´ , `)´ eta hauen artekoak, eta LA izan ezik. Guzti hauek ezabatu egiten dira ordezko karaktererik idatzi gabe.
- `(´ eta `)´-aren arteko sekuentziak kabia daitezke.

- Ixten ez diren parentesi irekiek edo alderantzizkoek (ireki gabe ixten direnek) programa amaiarazi egiten dute errore-mezu batez.
- Lerro bakoitza, gehienez ere, 80 karakterez osatzen da.

Honako hau espezifikazotzat onar daitekeen deskribapen zehatzagoa da.

Espezifikazio kontzeptuak zerikusi zuzena du abstrakzioarekin. Abstrakzioa programak modulutan banatzeko erabiltzen da, baina ulergaitz gertatzen da ez badago deskribatuko duen espezifikaziorik.

Datu edo sarrera-objektuen funtzio gisa uler daiteke programa bat, emaitza edo irteera-objektuak lortzeko helburuarekin. Programa bat abstrakzio funtzional baten inplementazioa da. Abstrakzio funtzional bat era desberdinetan inplementa daiteke, baina guztiek bete behar dute espezifikazioa. Hots, espezifikazioak abstrakzio funtzionala definitu edo deskribatzen du, eta inplementatzen duen programak, bestalde, funtzio hori kalkulatzeko modu bat definitzen du. Espezifikazioa, sarrera-objektu posibleen eta dagozkien irteera-objektuen arteko erlazioaren deskribapen zehatza da.

1.2. ZERTARAKOAK

Gaur egun espezifikazioa informatikako hamaika alorretan ageri da. Batetik, edozein programaren diseinurako premiazkoa da ebatzi beharreko problemaren ezagutza zehatza (anbiguetaterik gabea) edukitzea diseinuari ekin aurretik; izan ere, diseinatu ahala adabatzeak akats ugari sortzen bait du. Problema espezifikatzeak ideiak egituratzera behartzen gaitu eta aspektu nabariak agerian uztera bidenabar. Aspektu hauek, espezifikazioan ez bada, diseinuan agertzen dira edo, are okerrago, ez dira agertzen, eta orduan programa desegokia izango da, ez bait du nahi dugun problema ebatziko.

Talde-lanei dagokienez espezifikazioak komunikazio-disziplina eragin behar du. Modulutako banaketa sistemaren eginbeharrak zehatz-mehatz ezagutuz egin behar da eta modulu bakoitzeko diseinatzaileak, bai ebatzi beharreko problema, bai beste moduluekiko elkarrekintzak, inolako anbiguetaterik gabe ezagutu behar ditu.

Programen testak diseinatzeko ere ezinbesteoak dira espezifikazioak, bereziki "kutxa beltza" izeneko testetan.

Bestalde, ez dugu ahantzi behar programak behin egin eta askotan irakurtzen direla. Programa baten mantenimendurako beharrezkoa da dokumentazio aproposa edukitzea, edozeinek programa horrek zer egiten duen zehazki uler dezan aparteko ahaleginik gabe. Gerora ere, aldaketaren bat burutu beharko denean, ezinbestekoa izango da dokumentazioa. Kontutan hartu, aldaketa hauek egiten dituena eta hasierako diseinua egin zuena normalean ez direla pertsona bera izaten.

Programen egiaztapenerako ere espezifikazioak dira abiapuntua, azken finean programak egiaztatzea espezifikazioak betetzen dituztela formalki frogatzea bait da, hau da, diseinatu zeneko eginbeharrak betetzen dituela frogatzea. Beraz, espezifikazioak, egiaztapenean oinarritzen diren programen diseinu-metodologien abiapuntua ere badira.

Programen sintesia, espezifikazioan oinarrituz programak era automatikoz sortzean datza. Azkenik, espezifikazio-lengoaia exekutagarriak ere baditugu, hau da, programazio-lengoaia bilakatu diren espezifikazio-lengoaiak, nolabait problemaren deskribapena egikaritzen dela.

1.3. ESPEZIFIKAZIO-LENGOAIAK

Espezifikazioak hainbeste zereginetan erabiltzeak espezifikazio-lengoaien ugaltzea ekarri du: nolako helburua, halako idazkera aproposa. Edozein lengoaia definitzerakoan bi aspektu hauek hartu behar dira kontuan:

* Sintaxia: Zer idatz dezakegu?
* Semantika: Zer adierazten du idazten dugunak?

Lengoaia bat formaltzat hartuko dugu baldin eta bere sintaxia eta semantika formalki definituta badaude (zehaztasun matematikoz). Helburuak baldintzatuko du lengoaiaren formaltasun-maila: taldeen arteko komunikaziorako, test-diseinurako edo dokumentaziorako lengoaia ez oso formala nahikoa izan daiteke; baina egiaztapenerako, programen sintesirako edo espezifikazioen exekuziorako ezinbestekoa da lengoaia formala, espezifikazioak matematikoki eta/edo automatikoki tratatzen bait dira.

Espezifikazio formalek esanahi zehatza bermatzen dute. Aldiz, espezifikazio ez-formalak errazago idatzi edo irakurtzen dira, baina zaila da esanahi guztiz zehatza adieraztea.

Espezifikazio-lengoaia formalek nahiz ez-formalek ondoko ezaugarriak bete behar dituzte:

- Argitasuna: Espezifikazio ulerterrazak (lengoaia ezagutzen duen edonorentzat).
- Laburtasuna: Ahalik eta laburrenak, erredundantziarik gabekoak.
- Zehaztasuna: Anbiguetateak uxatzen dituztenak.
- Murriztasuna: Inplementazio onartezinak ametitzea ekiditen dutenak.
- Orokortasuna: Premiagabeko baldintzarik ez, espezifikazioek ez dute inplementazio onargarririk baztetu behar.

Azken bi ezaugarriek, abstrakzio funtzionalen funtsezko aspektuak juxtu-juxtu definitu behar direla adierazten dute, axolagabeko xehetasunak albo batera utziz.

1.3.1. Espezifikazio ez-formalak

Espezifikazio ez-formalerako teknika batek edozein abstrakzio funtzionalen aspektu nabariak klausulen bidez adierazteko formatoa finkatzen du. Moldea bai, baina klausulak adierazteko lengoaia formalik ez du ezartzen. Espezifikatzen duenaren esku geratzen da, hortaz, notazio matematiko eta/edo teknikoaren hautaketa, lengoaia naturalaren laguntzaz deskribapen argiak, laburrak eta zehatzak idatzi ahal izateko.

Ikus dezagun espezifikazio ez-formalerako teknika bat:

- espezifikazioa: < abstrakzio-funtzional-izena >
- datuak: < datuen-mota-eta-izena >
- emaitzak: < emaitzen-mota-eta-izena >
- aldatuak: < aldatu-diren-datuak >
- aurreko-baldintza: < datuen-murriztapenak >
- eragina: < portaeraren-deskribapena >
- salbuespenak: < aurrikusten-diren-salbuespen-kasuen-deskribapena >

Metodo honen ideia, batetik, sintaxia karakterizatzea da, datuetatik emaitzerako funtzio matematikoari abstrakzio funtzional bat egokituz, eta horretara daude zuzenduak, hain zuzen, lehen hiru klausulak. Bestetik, semantika ere karakterizatu behar da, hau da, sarrera eta irteeraren arteko erlazioa, eta hauxe da lau azken klausulen helburua.

Aurreko-baldintza klausulak funtzioaren eremua mugatzen du (funtzio partzialak) eta aldatuak klausularen bidez abstrakzio funtzionala eta funtzio matematikoa desberdintzen dira, abstrakzio funtzionalak aldatzen dituen datuak aipatuz. Azkeneko bi klausuletan abstrakzio funtzionalak normalki duen eragina eta salbuespen-kasuen arteko banaketa egiten da. Ikus ditzagun adibide batzuk, horietan nabarmentzen bait da klausulen egitekoa eta espezifikazioen ezaugarriak ondo betetzearen garrantzia.

1.1. *adibidea*: *Bi zenbaki arrunten arteko zatidura osoa kalkulatzen duen ekintza abstraktua espezifikatu.*

Enuntziatu hau irakurrita bi galdera nagusiri erantzun beharrean aurkitzen gara: Zatitzailea zeroa izan al daiteke? Emaitza non utziko dugu?

Horra aukeratu dugun erantzuna:

- espezifikazioa: ZATI
- datuak: x,y : osoak
- emaitzak: z : osoa
- aldatuak: (ez dago, honelakoetan klausula hau ez da jartzen)
- aurreko-baldintza: $x \geq 0$ eta $y > 0$
- eragina: $z = e$ zenbaki oso haundiena, non $e \leq x/y$
- salbuespenak: (salbuespen-kasurik ez dagoenez klausula hau idatzi beharrik ez dago)

Aurrekoaren bariantea, x eta y-ren arteko zatidura kalkulatu eta x-n emaitza uzten duena:
- espezifikazioa: ZATIM
- datuak: x,y : osoak
- emaitzak: x : osoa
- aldatuak: x
- aurreko-baldintza: $x \geq 0$ eta $y > 0$ eta $x = a$
- eragina: $x = a/y$

Nola erabili aldagai aldatuen hasierako balioak? Abstrakzioaren eragina adierazteko beharrezkoa da balio hauen erabilera finkatzea eta, horretarako, aukera bat baino gehiago proposatzen da.

Autore batzuek X_{aurre}, X_{ondo} edo X, X' bezalakoen bidez adierazten dituzte X aldagaiaren hasierako eta bukaerako balioak, hain zuzen notazio-arazoa besterik ez bait da. Guk erabiliko dugun notazioa aurreko baldintzan hasierako balio sinboliko bat ematean datza.

1.2. *adibidea*: *Array bat eta elementu bat emanda, elementu hori aurkitzen deneko posizioetako bat itzultzen duen ekintzaren espezifikazioa.*
- espezifikazioa: BILA
- datuak: A : array [1..n] osozkoa; t : osoa
- emaitzak: i : osoa
- aurreko-baldintza: $n \geq 1$
- eragina: $1 \leq i \leq n$ eta $A[i] = t$
- salbuespenak: i=0, baldin $A[j] \neq t$ edozein $1 \leq j \leq n$-rentzat

Hauxe da espezifikazio orokorrena, baina ez litzateke onargarria izango elementua ageri den lehenengo posizioa lortu nahi bagenu; kasu honetan murriztaileagoa egin beharko genuke:
- eragina: $1 \leq i \leq n$ eta $A[i] = t$ eta $A[j] \neq t$ edozein $1 \leq j < i$-rentzat

1.3. *adibidea*: *Osozko sekuentzia baten azken zenbaki negatiboa eta dagoen posizioa kalkulatzen duen abstrakzio funtzionala espezifikatu*:
- espezifikazioa: AZKENG
- datuak: $<e_1, e_2, ..., e_k>$: osoak
- emaitzak: i, x : osoak
- aurreko-baldintza: $k \geq 1$ ($k \geq 0$ sekuentzia hutsa onartuz gero)
- eragina: $1 \leq i \leq k$ eta $x = e_i < 0$ eta $e_{i+1}, ..., e_k \geq 0$
- salbuespenak: $i = 0$ eta $x = 0$, baldin $e_1, ..., e_k \geq 0$ (edo $k = 0$)

Parentesien bidez, lengoaia naturaleko enuntziatuan argi ez dauden aspektu desberdinak aukeratu ditzakegula adierazten da.

Orain arteko adibideek garbi asko erakutsi digute zein den espezifikazio ez--formalerako teknika honek eskatzen duen erabilera. Hala ere, enuntziatu labur eta ulerkor hauetan ez da benetan nabarmentzen espezifikazioaren komenientzia eta teknika honen interesa. Ondoko adibedeetan erraz konprenitzen da zenbateraino den inportantea notazio egokiz eta txukun espezifikatzea, enuntziatuak berak konplexuak edo ilunskak direlako, hain zuzen ere.

1.4. *adibidea*: *Jo dezagun ondokoa burutzen duen programa espezifikatu behar dugula: $ ikurraz banantzen diren N elementutako azpisekuentziez osatutako zenbaki osoen fitxategia emanda (azken azpisekuentziak N elementu baino gutxiago edukiko ditu) kalkulatu azpisekuentzien kopurua, azken azpisekuentziaren luzera eta elkarren segidan dauden zenbaki berdinen lehenengo bikotea eta, bestalde, azpisekuentzia bakoitzarentzat, ea azpisekuentziaren elementurik txikiena eta handienaren arteko zenbaki lehen guztiak dauden ala ez eta zero baten atzetik doazen elementu bikoitien kopurua*.

Espezifikatzen hasitakoan, dudarik ez, zalantzak sortuko litzaizkiguke:

- N zenbakia datu gisa hartu behar al da?
- $ ikurraz hasi eta bukatzen al da sekuentzia?
- Azpisekuentzia hutsak onartuko al dira? Eta azpisekuentziarik ez egotea?
- Elkarren segidako al dira $ ikurra tartean duten zenbakiak?
- Elkarren segidan dauden zenbaki berdinen bikoterik ez badago, zer egin behar du programak?

Galdera guzti horiei erantzun eta programaren eginbeharrak zehazten dituen espezifikazioa hauxe izan daiteke:

- espezifikazioa: AZPISEK
- datuak: N: osoa,
  S: $< \$ as_1 \$ as_2 \$ ... \$ as_k \$ >$, non
  $as_i = < e_{i1} , e_{i2} , ... , e_{iN} >$ edozein $1 \leq i < k$-rentzat
  $as_k = < e_{k1} , e_{k2} , ... , e_{kL} >$
  e_{iz} : osoak
- emaitzak: zenbatas, luzazk: osoak,
  bikote: osoak x osoak,
  $n_1, n_2, ... , n_k$: osoak,
  $b_1, b_2, ... , b_k$: boolearrak,
- aurreko-baldintza: $k \geq 0$ eta $0 \leq L < N$ eta $N > 0$

- eragina:

  zenbatas = k
  luzazk = L
  bikote = (e_{iZ} , e_{iZ+1}) non $e_{iZ} = e_{iZ+1}$ eta
  $e_{pm} \neq e_{pm+1}$ honakoetan:
  $p < i$ eta $1 \leq m < n$
  edo
  $p = i$ eta $1 < m < z$

  Edozein $1 \leq i \leq k$-rentzat:
  n_i = kard { (e_{iZ} , e_{iZ+1}) / $e_{iZ} = 0$ eta e_{iZ+1} bikoitia da }

  b_i = true b.s.b. $as_i \supseteq$ { x / x lehena da eta min $(as_i) \leq x \leq$ max (as_i) }
- salbuespenak:
  baldin $e_{pm} \neq e_{pm+1}$ bada edozein p,m-rentzat, orduan:
  bikote = (0 , 1)

1.5. *adibidea*: *Demagun damen antzeko joko batean ari diren bi jokalarien iharduera gestionatuko duen programa bat diseinatu behar dugula. Bi jokalariek ordenadorea aukeratu dute euskarri bezala, teklatutik adierazten dituzte beren mugimenduak eta pantailaz baliatzen dira unean-uneko jokoaren egoera ikusteko. Damen antzeko jokoa diogu, izan ere ia-ia dametan ari bait dira baina bi mugapen ezarri dituzte: ez da derrigorrezkoa jatea eta aurkako eremuaren azken muturrera iristen diren fitxei ez zaie mugimendu-ahalmen berezirik ematen.*

Delako programak, jokoaren gestioa behar bezala burutu nahi badu, hainbat gauza eduki beharko ditu kontuan: nori dagokion jokatzea, jokaldien irakurketa, ea jokaldiak legezkoak diren, ...

Lehenbiziko hurbilpen gisa honako algoritmo abstraktu hau eman genezake:

```
begin
    HASIERAKO_EGOERA (e);
    MARRAZTU_EGOERA (e);
    HASIERAKO_TXANDA (tx);
    while not JOKOAREN_BUKAERA (e) do
        begin
            IRAKUR_JOKALDIA (jok);
            if LEGEZKO_JOKALDIA (e, jok, tx) then
                begin
                    JOKALDIA_BURUTU (e, jok, tx);
                    TXANDA_ALDATU (tx);
                    MARRAZTU_EGOERA (e)
                end
            else IDATZ_MEZUA (tx)

    end;
    IRABAZLEA (e)
end.
```

Algoritmo honetan e: EGOERA, jok: JOKALDI eta tx:TXANDA aldagaiak erabili dira, oraindik mota horiek espezifikatu gabe badaude ere.

Diseinuaren jarraipenak ekintza abstraktuak birfintzea eskatzen du, baina hori baino lehenago ezinbestekoa da ekintza horiek espezifikatzea. Bestela, zer egiten duten zehaztasun handiz jakin gabe, nola birfinduko ditugu? Egia da ekintzen izenek nola edo hala eginkizunaren ideia iradokitzen dutela, baina hori ez da nahikoa.

Espezifikatzera joko dugu, beraz, baina horretarako beharrezkoa da erabilitako datu-motei buruz zerbait gehiago jakitea.

EGOERA motako aldagaiak 8x8 matrizeak izango dira, eta bertan taulatuko laukitxo guztien berri jasoko da, hau da, e_{iZ} bakoitzak ($1 \le i,z \le 8$) 'T', 'B' edo ' ' balioak har ditzake, (i,z) laukitxoan fitxa txuria, beltza edo fitxarik ez dagoela adierazteko. EGOERA datu-mota beste zenbait informazioz ere aberastu daiteke, esate baterako, jokalari bakoitzak zenbat fitxa dauzkan edo zenbat jokaldi pasa diren elkarri jan gabe adieraziz. Aspektu hauek beharrezko gerta daitezke hainbanaketa noiz gertatzen den antzemateko.

JOKALDIA = LAUKITXO x LAUKITXO izango da, lehenengoa jatorrizkoa eta bigarrena helburu-laukitxoa. Halaber, LAUKITXO = OSOA x OSOA izango da, errenkada eta zutabea adieraziz hurrenez hurren.

TXANDA = ('T', 'B') hartuko dugu.

Ikus ditzagun, bada, HASIERAKO_EGOERA eta LEGEZKO_JOKALDIA ekintzen espezifikazioak:

- espezifikazioa: HASIERAKO_EGOERA
- emaitzak: e:EGOERA
- eragina:

e_{iZ} = 'T' baldin eta ondorengo kasuren bat gertatzen bada:

a) (i = 1 edo i = 3) eta z bakoitia, $1 \leq z \leq 8$

b) i = 2 eta z bikoitia, $1 \leq z \leq 8$

eta

e_{iZ} = 'B' baldin eta ondorengo kasuren bat gertatzen bada:

a) (i = 6 edo i = 8) eta z bakoitia, $1 \leq z \leq 8$

b) i = 7 eta z bikoitia, $1 \leq z \leq 8$

eta

e_{iZ} = ' ' gainontzeko kasuetan.

- espezifikazioa: LEGEZKO_JOKALDIA
- datuak:

$$e = \begin{pmatrix} e_{11},\ e_{12},\ \ldots,\ e_{18} \\ e_{21},\ e_{22},\ \ldots,\ e_{28} \\ \ldots,\ \ldots,\ \ldots,\ \ldots,\ \ldots \\ e_{81},\ e_{82},\ \ldots,\ e_{88} \end{pmatrix} : \text{EGOERA}$$

jok = ((x,y), (u,z)) : JOKALDIA

tx : TXANDA

- emaitzak: b:boolearra
- aurreko-baldintza: $1 \le x, y, u, z \le 8$
- eragina:

b = true b.s.b.

1) $e_{xy} = tx$ eta $e_{uz} = ' '$ eta

2) ondoko kasuren bat gertatzen da:

a)

$$(u,z) = \left\{ \begin{array}{l} \left\{ \begin{array}{c} (x+1, y-1) \\ \text{edo} \\ (x+1, y+1) \end{array} \right\} \text{ baldin } tx = 'T' \\ \left\{ \begin{array}{c} (x-1, y-1) \\ \text{edo} \\ (x-1, y+1) \end{array} \right\} \text{ baldin } tx = 'B' \end{array} \right\}$$

b) badago era honetako laukitxo-sekuentzia bat:

$< (u_1, z_1), (u_2, z_2), \ldots, (u_n, z_n) >$, non:

- $n \ge 2$ eta $(u_1, z_1) = (x,y)$ eta $(u_n, z_n) = (u,z)$
- baldin tx = 'T', orduan edozein $1 \le i < n$-rentzat:

  $(u_{i+1}, z_{i+1}) = (u_i+2, z_i-2)$ eta $e_{u_i+1, z_i-1} = 'B'$

  edo

  $(u_{i+1}, z_{i+1}) = (u_i+2, z_i+2)$ eta $e_{u_i+1, z_i+1} = 'B'$
- baldin tx = 'B', orduan edozein $1 \le i < n$-rentzat:

  $(u_{i+1}, z_{i+1}) = (u_i-2, z_i-2)$ eta $e_{u_i-1, z_i-1} = 'T'$

  edo

  $(u_{i+1}, z_{i+1}) = (u_i-2, z_i+2)$ eta $e_{u_i-1, z_i+1} = 'T'$

1.3.2. Aurre-ondoetako espezifikazio formala

Programen egiaztapena eta eratorpen formalerako ezinbestekoa da espezifikazio formalak erabiltzea. Aurre-ondoetako espezifikazioa aurreko baldintza eta ondoko baldintza izeneko bi formulez osatzen da. $\{\phi\}$ P $\{\psi\}$ notazioak, non ϕ aurreko baldintza, P programa eta ψ ondoko baldintza diren, P programak ϕ eta ψ formulen bidez adierazitako espezifikazioa betetzen duela baieztatzen du, hau da, P exekutatu aurretik ϕ egiazkoa bada, orduan exekutatu ondoren ψ ere egiazkoa izango dela. Esate baterako, $\{y \neq 0\}$ P $\{z = x/y\}$ notazioaren bidez hauxe adierazten da: hasieran y aldagaiak zero balioa ez duen guztietan, P programa exekutatzearen ondorioz z-n x eta y-ren arteko zatidura edukiko dugu. Baina, aurreko eta ondoko baldintzak idatzita dauden lengoaiaren arabera aurre-ondoetako espezifikazioa formala edo ez-formala izan daiteke. Gure helburuetarako lengoaia formala hautatu dugu: lehen mailako logikaren lengoaia. Aurreko baldintza formularen aldagaiak programaren sarrerako datuak dira, nolabait bete beharreko baldintzak definitzen dituelarik. Ondoko baldintzak datu eta emaitzen arteko erlazioa finkatzen du, programaren esanahia agerian utziz.

Beraz, formula bion bidez objektuen propietateak adierazten dira. Ezin ahantzi, hala ere, objektu hauek datu-mota batekoak edo bestekoak direla eta, horregatik, erabiliko dugun lengoaia ikusi aurretik, Pascal-eko datu-moten erabilera formalari erreparatuko diogu, notazioak behar bezala finka ditzagun.

Ariketak

1.1.- Espezifikatu ez-formalki ondorengo abstrakzio funtzionalak:

- Erabaki ea zenbaki oso bat kapikua den.

- Emandako bi array-en elementu errepikatuak lortu.

- Zenbaki osoen multzo bat bi azpimultzo disjuntutan banatu, non azpimultzo hauetako elementuen baturak berdinak izatea bete behar bait da.

- Input-etik zenbaki osoen sekuentzia bat irakurri eta output-ean idazten du. Abstrakzio honek input-eko zenbakiak goranzko ordenan idazten ditu, lerro bakoitzean zenbaki bana eta, ondoren, zenbaki hori input-ean zegoeneko posizioa. Zenbaki errepikatuak daudenean, dagozkien posizioak goranzko ordenan idazten dira. Adibide honetan sarrera konkretu bati dagokion irteera ikus daiteke:

  input = < 3, 4, 7, 3, 2 >
  output = < < 2, 5 >, < 3, 1 >, < 3, 4 >, < 4, 2 >, < 7, 3 > >

- S[1..n] eta I[1..n] array-etan n koloretako konbinazioak gordetzen dira (errepikapenik gabe), koloreak hauexek izan daitezkeela: gorria, urdina, horia, txuria, beltza, berdea eta marroia. S eta I emanda, a) eta b) emaitzak lortzen ditu abstrakzio funtzional honek, non:
  a) I-n eta S-n egon bai, baina posizio desberdinetan dauden koloreen kopurua,
  b) I-n eta S-n egoteaz gain, posizio berdinetan dauden koloreen kopurua.
  Adibidez:
  S=(gorria, marroia, horia, berdea, urdina)
  eta
  I=(berdea, marroia, txuria, beltza, gorria) edukiz gero,
  a) partean lortutako kopurua 2 litzateke (berdea eta gorria koloreengatik) eta b)n, berriz, 1 (marroiarengatik).

- A[1..n] array-an n koloretako konbinazio bat daukagu, kolore gisa gorria, txuria eta urdina onartzen direlarik. A emanda, array berean hasierako koloreen konbinazio ordenatua (gorri guztiak hasieran, txuriak erdian eta urdinak bukaeran) lortzen du ekintza abstraktu honek.

- N erreginen problema ebatzi: "n x n jake-taulan n erregina kokatu elkar jan gabe".
- Luzera berdineko azpisekuentziez osatutako zenbaki osoen fitxategia emanda, non fitxategiaren lehen elementuak azpisekuentzi kopurua adierazten duen eta bigarrenak hauen luzera, kalkulatu:

  a) Fitxategi osoarentzat:
  - Gutxienez 0 bat daukaten azpisekuentzien kopurua.
  - Azpisekuentzi kopurua bikoitia bada, azken azpisekuentziaren elementuen batura.

  b) Azpisekuentzia bakoitzarentzat:
  - Azpisekuentziako zenbaki lehen handiena, baldin eta azpisekuentziako maximoa ere bada.

- Sarrerako fitxategiaren testuko bokalen agerpen-maiztasuna kalkulatu.

2. DATU-MOTEN TRATAMENDU FORMALA

Datu-mota, balio-multzoa eta multzo horren gaineko eragiketa-multzoa da. **Balioa**, izatez, ezaugarri denboral edo espazialik ez duen abstrakzio matematikoa da eta, horregatixe, aldaezina da, nahiz eta ordenadorearen memorian errepresenta daitekeen. Objektuak, aldiz, ezaugarri denboral eta espazialak baditu eta programan zehar indefinituta egon edo balioren bat eduki dezake. Objektuak balio bat edukitzeak balio horren errepresentazioa duela esan nahi du, ezinbestean memoriako espazioa hartuz. **Konstantea**, bere balioa aldaezina den objektua da. **Aldagaia**, ordea, balioa alda dezakeen objektua da. **Eragiketa** balioen funtzio matematikotzat har daiteke. Eragiketa-sinbolo edo eragile batek eragiketa zehatz bat adierazten du. Horrela osatutako adierazpenak ere objektuak dira.

Datu-mota batek ondokoak biltzen ditu:

- Balio-multzoa: Mota horretako objektuek har ditzaketen balioen multzoa (motaren heina). h(T) notazioaren bidez adieraziko dugu T motaren heina.
- Eragiketa-multzoa: Motako balio-multzoan eremua eta/edo heina duten eragiketen multzoa (motari dagozkion eragiketak).

Datu-moten erabilpenak honako abantail hauek dakartza:

- Objektuei buruzko ideiak argitu eta egituratzen laguntzen dio erabiltzaileari.
- Konpilazioaldiko moten gaineko kontrolaren inguruan: Programaren testua aztertze soilarekin aski du konpiladoreak objektuen gaineko eragiketak motekiko koherenteak diren egiaztatzeko.
- Objektuen eta moten erazagupena kontuan hartuz, programaren exekuziorako behar den memoria finka dezake konpiladoreak.

Datu-motaren kontzeptuak ere badu zerikusirik abstrakzioarekin, datu-mota bakoitzeko bi maila bereiz daitezke eta:

- Maila abstraktua: Balioen heina eta dagozkion eragiketak.
- Inplementazio-maila: Balioen memoriako errepresentazioa eta eragiketen inplementaziorako algoritmoak.

Aldagai baten datu-motak bere balio guztien ezaugarri komunak finkatzen ditu, (barne-errepresentazioari erreparatu gabe) oinarrizko eragiketen propietateen bidez (hauen inplementazioak ere kontuan hartu gabe). Axioma-multzoen bitartez adierazi ohi diren oinarrizko eragiketen propietateek, inplizituki, balioak karakterizatzen dituzte, ezin bait dira erabili eragiketa hauen bidez ez bada.

Goi-mailako lengoaietan gehien erabiltzen diren datu-motek eredu matematiko ezagunekin lotura estua dute eta, horregatik, beraien propietate matematikoak ederki mugatuta daude. Esate baterako, Osoak, Errealak, Biderkaketa Cartesiarra, Multzoak edo Sekuentziak bezalako motak nahiko adierazgarriak dira.

Pascal izan zen kontzeptuok honela erabili zituen lehenengoetako lengoaia, aurretik datu-motak balio-multzoak besterik ez bait ziren.

Pascal-ez datu-mota sinple eta egituratuak bereizten dira. Datu-mota sinpleak, honakoak: Integer, Real, Char eta Boolean. Datu-mota hauetako objektu eta eragiketen notazioa, formalismoa eta propietateak, bakoitzari egokitzen zaion eredu matematikoaren araberakoak dira. Hots, Integer-entzat ($\mathbb{Z}$, +, *, div, /, -, old, ...) eta Real-entzat ($\mathbb{R}$, +, *, /, ...), Char-entzat ({`a´,..., `z´, `0´, ..., `9´, ` ´, ...} ord, suc, pred,...) eta Boolean-entzat Booleren algebra ({T, F}, not, and, or, ...). Eredu hauek dagoeneko ezagunak ditugu.

Datu-mota egituratuetako objektuak, objektu sinpleagoen bildumez (normalean osagaiak deitutakoak) eta atzipenerako nahiz aldaketarako oinarrizko eragiketez osatzen dira, era batean edo bestean. Pascal-ez lau modu desberdinetan konposatzen dira aldagai sinpleak, bakoitzari kontzeptu matematiko desberdina dagokiola.

2.1. SET (MULTZOA)

Pascal-eko set motako objektuak multzoak dira. Zehazkiago, *type M = set of T* erazagupenak M datu-mota definitzen du, eta

$$h(M) = \wp(h(T)), \text{ non}$$

T oinarrizko mota bait da eta $\wp$(A)k A multzoaren parteen multzoa denotatzen bait du.

Adibidea: *type KL = set of `a´..`c´* erazagutuz gero, KLren heina = $\wp$ ({`a´,`b´,`c´})

Beraz, edozein *var x: KL* aldagaik $\wp$ ({`a´,`b´,`c´}) honetako edozein multzo har lezake baliotzat, hau da, multzo hutsa edo `a´,`b´ eta `c´ karaktere guztiez edo batzuez osatutako edozein multzo.

Mota honetako eragiketei multzoetako eragiketa matematikoak dagozkie, sintaxia aldatu besterik gabe.

Pascal	Matematikak	
in	$\in$	(barnekotasuna)
+	$\cup$	(bilketa)
*	$\cap$	(ebaketa)
-	$-$	(kenketa)

2.2. ARRAY

Array mota bi ikuspegi desberdinetatik formaliza daiteke:

a.- Multzo baten eta bere buruaren arteko biderkadura Cartesiarra.

A x B biderkadura Cartesiarra honela definitzen da:

$(a,b) \in AxB \leftrightarrow a \in A$ eta $b \in B$

eta bere elementuen arteko berdintza:

$(a,b) = (c,d) \leftrightarrow a = c$ eta $b = d$

Hortaz, $M^n = \underbrace{M \times M \times \ldots \times M}_{n}$ multzoa, M multzoko elementuen n-koteez osatutako multzoa, ondokoa besterik ez da:

$(m_1, m_2, ..., m_n) \in M^n \leftrightarrow m_i \in M$ edozein $1 \leq i \leq$ n-rentzat

eta bere elementuen arteko berdintasuna:

$(m_1, m_2, ..., m_n) = (d_1, d_2, ..., d_p) \leftrightarrow n = p$ eta $m_i = d_i$ edozein $1 \leq i \leq$ n-rentzat

Bada, *type A = array [1..n] of T* definizioak, oinarrizko mota T dela, A datu-mota definitzen du, non motaren heina hauxe bait da:

$h(A) = h(T)^n = \{(t_1, ..., t_n) / t_i \in h(T)$ edozein $1 \leq i \leq$ n-rentzat$\}$, hau da, *var a : A* aldagaiak $h(T)^n$ren barneko edozein $(a_1, ..., a_n)$ n-kote eduki dezake balio gisa. Eragiketei buruz, berriz, A motari n eragile dagozkio, $1 \leq i \leq n$ balio guztientzat bana:

$[i] : h(T)^n \rightarrow h(T)$, non $[i]((a_1, ..., a_n)) = a_i$

eta Pascal-ez [i] (a) denotatzeko a[i] idazten da.

b.- Indize-multzoan eremua duten funtzioak.

Array-ak, bestalde, indize-multzotik oinarrizko motarako funtzioak bezala ere formaliza daitezke. *type A = array [1..n] of T* emanez gero, A datu-motaren heina funtzio-multzo hau litzateke: $\{a: \{1..n\} \rightarrow h(T)\}$. Kasu honetan, motako eragiketak eremuen gaineko array-ak berak dira, Pascal-eko a[i] notazioak eremuko i balioaren funtzioa adierazten duelarik, hau da, a(i) adierazpena. Hari beretik, funtzio-konposaketa ere definituta dago eta Pascal-eko a[b[i]] adierazpenak $(a(b(i)) = a \circ b(i)$ konposaketa denotatzen du.

2.3. RECORD (ERREGISTROA)

Erregistroei atxikitutako eredu matematikoa ere biderkadura Cartesiarra da, baina multzo desberdinen artekoa.

```
type    R =  record
                s1 : T1 ;
                …
                sn : Tn
             end;
```

sententziaren bidez definitutako R motaren heina $h(T_1)$ x ... x $h(T_n)$ren n-koteez osatutako multzoa da:

$h(R) = h(T_1)$ x...x $h(T_n) = \{(t_1, ..., t_n) / t_i \in h(T_i)$ edozein $1 \le i \le n$-rentzat$\}$

Mota honi n eragiketa dagozkio, $1 \le i \le n$ bakoitzarentzat bana:

$S_i : h(T_1)$ x ... x $h(T_n) \rightarrow h(T_i)$ non,

$t = (t_1,...,t_n) \in h(T_1)$ x ... x $h(T_n)$ emanik, orduan $S_i(t) = t_i$, Pascal-eko puntu-notazioaz honela adieraziko litzatekeena: $t . s_i$.

Pascal-ez badaude erregistro aldagarriak ere, eta berauen barne-kontzeptu matematikoa $T_1,...,T_n$ multzoen bildura disjuntua da. Har dezagun:

```
type      RA2 =   record
                    case  tag : (k1, k2)  of
                                  k1 : (s1: T1);
                                  k2 : (s2: T2)
                    end;
```

non k_1, k_2 konstante-identifikadoreak bait dira, eta RA2ren heina T_1 eta T_2ren arteko bildura disjuntua bait da, honela: $\{k_1\}$ x $h(T_1) \cup \{k_2\}$ x $h(T_2)$

Orokorrean:

```
type      RA  =  record
                    case tag : (k1,...,kn) of
                                  k1 : (S1,1: T1,1 ,..., S1,m1 : T1,m1);
                                  …
                                  kn : (Sn,1: Tn,1 ,..., Sn,mn: Tn,mn)
                    end;
```

eta definizio honen heina honako bildura disjuntua da:

$\{k_1\}$ x $h(T_{1,1})$ x ... x $h(T_{1,m_1}) \cup ... \cup \{k_n\}$ x $h(T_{n,1})$ x ... x $h(T_{n,m_n})$

Hortaz, *var t: RA* erazagutuz gero t-ren balioa edozein $(t_o,...,t_p)$ p-kote izan liteke, non $t_o = k_i$ izango den $1 \le i \le n$ batentzat eta, halabeharrez, $p = m_i$. Hauetako i bakoitzari m_i+1 eragiketa dagozkio:

- tag eragiketa, non $tag(t) = t_o$ bait da, Pascal-ez t.tag adierazpenaren bidez denotatzen dena.
- $S_{i,j}$ eragiketak (guztira m_i), $1 \le j \le m_i$ bakoitzarentzat bana, non $S_{i,j}$ funtzio partziala den:

$S_{i,j}(t) : h(RA) \rightarrow h(T_{1,1}) \cup ... \cup h(T_{1,m_1}) \cup ... \cup h(T_{n,1}) \cup ... \cup h(T_{n,m_n})$

$$S_{i,j}(t) = \begin{cases} t_j & \text{baldin } t_0 = k_i \\ \text{indefinitua} & \text{bestela} \end{cases}$$

Pascal-ez $S_{i,j}(t)$ denotatzeko $t.s_{i,j}$ erabiltzen da.

2.4. FILE (FITXATEGIA)

Fitxategien barne-kontzeptua sekuentzia da, biderkadura Cartesiarraren bidez honela defini daitekeena:

T multzoa eta n zenbaki arrunta emanda, n aldiz egindako T eta bere buruaren arteko biderkadura Cartesiarrari T^n deituko diogu, hau da, $T^n = \underbrace{T \times T \times .. \times T}_{n}$, non $T^o = \{< >\}$ eta $T^1 = T$ kasu partikularrak diren. Gatozen definitzera:

a) T* multzoa, Tko elementuen sekuentzien multzoa:

$$T^* = T^o \cup T^1 \cup ... \cup T^n \cup ...$$

eta T^+, Tko elementuen sekuentzia ez-huts guztien multzoa:

$$T^+ = T^1 \cup T^2 \cup ...$$

b) Sekuentzi kateamendua: T*-ren barneko $< a_1,..., a_n >$ eta $< b_1,..., b_m >$ sekuentziak emanda, bi sekuentzion kateamendua, $< a_1,..., a_n > \bullet < b_1,..., b_m >$ jarriz adieraziko duguna, $< a_1,..., a_n, b_1,..., b_m > \in T^*$ sekuentzia da.

c) Sekuentzia ez-huts baten lehena:

lehen : $T^+ \rightarrow T$, non

lehen $(< a_1,..., a_n >) = a_1$

d) Sekuentzia ez-huts baten hondarra:

hondar : $T^+ \rightarrow T^*$,

non, hondar $(< a_1, a_2, ..., a_n >) = < a_2, ..., a_n >$.

Hots, edozein $s \in T^+$-rentzat

$<$lehen (s)$> \bullet$ hondar (s) = s izango da.

"Type F = file of T" definituz gero F datu-motaren heina h(F) = h(T)* da. Aurreko datu-motetan ez bezala, ez dago zuzeneko atzipenik, eragiketa bakarra atzipen sekuentziala da eta. Elementu bakar bat atzi daiteke aldiberean. Atzipen sekuentziala formalizatu ahal izateko, beharrezkoa da fitxategi motako objektuaren egoera kontutan hartzea, alegia, ez dela sekuentzia motako balioa bakarrik ezagutu behar, baizik eta une bakoitzean atzigarria den sekuentziaren elementua ere bai. $\overleftarrow{f} \bullet \vec{f}$ notazioaz $f \in T^*$ adieraziko dugu, $f = \overleftarrow{f} \bullet \vec{f}$, non $f\uparrow$ = lehen $(\vec{f})$ une bakoitzeko elementu atzigarria izango den. Horregatik, fitxategien gaineko oinarrizko eragiketak, $\vec{f}$, $\overleftarrow{f}$ eta $f\uparrow$ aldiberean aldatzen dituzten eragiketen baliokideak dira. Baliokidetasun hauek, aldibereko asignazioen bidez azalduko ditugu:

reset (f) = [$\vec{f} := \overleftarrow{f} \bullet \vec{f}$
$\overleftarrow{f} := <>$
$f\uparrow := lehen\ (\overleftarrow{f} \bullet \vec{f})$]

get (f) = [$\overleftarrow{f} := \overleftarrow{f} \bullet <lehen\ (\vec{f})>$
$\vec{f} := hondar\ (\vec{f})$
$f\uparrow := lehen\ (hondar\ (\vec{f}))$]

put (f) = [$\overleftarrow{f} := \overleftarrow{f} \bullet <f\uparrow>$
$f\uparrow := indefinitua$
$\vec{f} := <>$]

rewrite (f) = [$\overleftarrow{f} := <>$
$\vec{f} := <>$]

eof (f) $\leftrightarrow$ [$\vec{f} = <>$]

Ariketa gisa interesgarria litzateke lerro-egitura duten fitxategiak formalizatzea sekuentziaz osatutako sekuentziak bezala (lerro bakoitza karaktere-sekuentzia da, eta fitxategia lerro-sekuentzia).

3. LEHEN MAILAKO LOGIKAREN LENGOAIA

Lengoaia formal bat definitzeko bere sintaxia eta semantika mugatu behar dira. Lehen mailako logika aztertzeko, lengoaiaren sintaxia aldagai, funtzio eta predikatu sinboloez osatutako alfabeto generiko baten gainean definituta eduki behar da. Lengoaia hau Pascal-ez idatzitako programen zuzentasunaz arrazonatzeko erabiliko dugunez, horretara egokitutako alfabetoa finkatuko dugu:

- Aldagai-sinboloak edozein motatako aldagai-identifikadoreak dira:
  A, f, input, f↑, x, y, z, aurre, zenb
- Konstante-sinboloak:
  3, `a´, < 5,7 >, < >, (3,2,1)
- Funtzio-sinboloak Pascal-eko funtzio standard guztien ikurrak edo dagozkien ikur matematikoak, "suc, "*", "div", "mod", "∪", "+", "round", e.a., dira. Notazio matematiko edo laburdura gisa erabili ohi diren funtzioak ere onartuko ditugu: ∑, !, e.a.
- Predikatu-sinboloak Pascal-eko funtzio boolear guztiak dira ("odd", ≤, ≥, ∈, ...) eta, bereziki, berdintasuna (=). Era berean, matematiketako edozein predikatu edo laburdura ere erabil daiteke, ∉ esate baterako.

Funtzio- eta predikatu-sinbolo bakoitzari anizkotasun jakin bat datxekio, behar dituen eragigaien kopurua adierazten duena.

Zenbat eta zabalagoa izan onartzen den alfabetoa, orduan eta malguagoa eta aberatsagoa da lengoaia, eta errazago gertatzen da beraren bidez propietateak adieraztea.

Hauxe da, hain zuzen ere, notazio matematiko hauek onartzearen arrazoia.

Alfabetoa finkatu ondoren, gatozen lengoaiaren sintaxia definitzera.

3.1. LENGOAIAREN SINTAXIA

Gaiak:

a) Edozein konstante gaia da,
b) Edozein aldagai gaitzat hartzen da, eta
c) $t_1,...,t_n$ gaiak badira eta f funtzio-sinbolo n-tarra bada, f $(t_1,...,t_n)$ ere gaia izango da.
   Gaien eta funtzio-sinboloen sintaxiaren arteko komunztadura beharrezkoa da, bestela, sortutako gaiak ez bait lirateke onargarriak izango.

Formulak:

d) $t_1,...,t_n$ gaiak badira eta P predikatu n-tarra, orduan $P(t_1,...,t_n)$ formula (atomikoa) da. Aipagarria da "=" berdintasun predikatu bitarra, maiz erabiliko dugu eta.

e) ϕ eta ψ formulak badira, $\neg(\phi)$, $(\phi \wedge \psi)$, $(\phi \vee \psi)$, $(\phi \rightarrow \psi)$, $(\phi \leftrightarrow \psi)$ ere formulak dira.

f) x aldagaia eta ϕ formula bada, $\forall x(\phi)$ eta $\exists x(\phi)$ formulak dira, non $\forall$ eta $\exists$ kuantifikatzaile unibertsala eta existentziala diren hurrenez hurren.

Formuletako aldagaien artean bi motatakoak bereiziko ditugu, segun-eta nolakoak diren beren agerpenak:

ϕ formulan x aldagaiaren agerpena librea dela esango dugu baldin eta ez badago inongo kuantifikatzaileren eraginpean, eta agerpena atxekia izango da bestelako kasuetan. Aldagai batek agerpen libreak eta atxekiak eduki ditzake formula berean. Aldagai bat librea da formula batean, gutxienez agerpen libre bat badauka formula horretan, eta bestela atxekia izango da.

3.1. *adibidea*: *Ondokoak lehen mailako formulak dira:*

a) $\forall x\ (0<x \rightarrow 3*x>z) \wedge \exists z\ (z>x)$

Formula honetan x-ren lehenengo agerpena atxekia da eta hirugarrena librea, eta z-ren lehenengoa librea eta bigarrena atxekia; hortaz, bai x, bai z aldagai libreak dira.

b) $\exists z\ (z>0 \wedge z<x)$

Kasu honetan z-ren bi agerpenak atxekiak dira eta x-rena, berriz, librea. Beraz, z aldagai atxekia da eta x librea.

3.2. LENGOAIAREN SEMANTIKA

Lengoaiaren semantikak gaien eta formulen esanahia definitzen du. Gaiek eta formulek esanahirik eduki dezaten, ezinbestekoa da aldagaiek balio zehatzak hartzea. Balio hauek, landuko dugun alorrean, programaren exekuzioko instant konkretu batean aldagaiek dituztenak dira, konputazio-egoerari dagozkionak, alegia. Horregatik semantikaren formalizazioari ekin aurretik konputazio-egoera modu zehatzago batez definituko dugu.

S konputazio-egoera funtzio bat da, non:

. Eremua aldagai-identifikadoreen multzoa den,

. Heina aldagai guztien moten heinen bildura den eta

. Aldagai bakoitzari bere motako balio bat egokitzen zaion.

Egoera funtzioa bikote-multzo batez adieraziko dugu aurrerantzean, bikote bakoitzak aldagai-identifikadore bat eta honi egokitutako balioa dituela.

3.2. *adibidea*: *Ondorengo programak a eta b zenbaki oso ez-negatiboen batura kalkulatzen du, baten gehiketa eta baten kenketaz aparteko eragiketarik erabili gabe:*

```
begin
    x: = a;  y : = b;
    while   y > 0   do
      begin    (*)
        x:=x+1;
        y:=y-1
      end
    end
```

Programa honen bukaeran gerta litezkeen egoera batzuk hauek lirateke:

{(a,7), (b,3), (x,10), (y,0)}
{(a,0), (b,5), (x,5), (y,0)}

(*) seinaleaz markatutako kontrol-puntuko egoera posible batzuk:

{(a,7), (b,3), (x,8), (y,2)}
{(a,7), (b,3), (x,9), (y,1)}

Ondorengoak, aitzitik, (*) puntuan gerta ezinezkoak dira:

{(a,7), (b,3), (x,6), (y,4)}
{(a,7), (b,3), (x,8), (y,4)}
{(a,7), (b,3), (x,9), (y,1)}

Egoera aldatu ere egin daiteke, bai (aldagaia, balioa) bikote berri bat erantsiz, bai aldagai bati balio zaharraren ordez beste berri bat egokituz. Edozein modutan ere, S egoeran x aldagaiari b balioa egokituz sortzen den egoera berria S[b/x] notazioaz adieraziko dugu.

Lehen mailako formula bat, ϕ, S egoeran ondo definituta dagoela esaten da, baldin eta ϕ-ko aldagai libreen identifikadore bakoitzari Sn bere motako balioren bat badagokio. Gaien eta formulen balioztapena soilik ondo definituta dauden egoeretan burutu daiteke. Ikus dezagun, bada, gaiak eta formulak nola balioztatzen diren.

Gaien balioztapena S egoeran

a) S (konstantea) = konstantea.
b) S (aldagaia) = S egoeran aldagaiari dagokion balioa.
c) S (f (t_1, ..., t_n)) = f (S (t_1), ...,S (t_n))

Formulen balioztapena S egoeran

$$\text{d) } S\left(P\left(t_1, t_2, \cdots, t_n\right)\right) = \begin{Bmatrix} \text{true} & \text{baldin } P\left(S\left(t_1\right), \cdots, S\left(t_n\right)\right) = \text{true} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

Egin kontu konstanteak, funtzioak eta predikatuak ez direla interpretatzen, ikur interpretagarriak izan ordez, aldez aurretiko interpretazio standard ezaguna bait dute.

e)

$$S(\neg\phi) = \begin{Bmatrix} \text{true} & \text{baldin } S(\phi) = \text{false} \\ \text{false} & \text{baldin } S(\phi) = \text{true} \end{Bmatrix}$$

$$S(\phi \wedge \psi) = \begin{Bmatrix} \text{true} & \text{baldin } S(\phi) = S(\psi) = \text{true} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

$$S(\phi \vee \psi) = \begin{Bmatrix} \text{true} & \text{baldin } S(\phi) = \text{true edo } S(\psi) = \text{true} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

$$S(\phi \rightarrow \psi) = \begin{Bmatrix} \text{true} & \text{baldin } S(\phi) = \text{false edo } S(\psi) = \text{true} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

$$S(\phi \leftrightarrow \psi) = \begin{Bmatrix} \text{true} & \text{baldin } S(\phi) = S(\psi) \\ \text{false} & \text{bestela} \end{Bmatrix}$$

f)

$$S(\exists x\phi) = \begin{Bmatrix} \text{true} & \text{baldin x – ren motako a balioren bat existitzen bada, non} \\ & S(a/x)(\phi) = \text{true} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

$$S(\forall x\phi) = \begin{Bmatrix} \text{true} & \text{baldin x – ren motako edozein a baliorentzat} \\ & S(a/x)(\phi) = \text{true betetzen bada} \\ \text{false} & \text{bestela} \end{Bmatrix}$$

3.3. *adibidea*: *Adibide honetako edozein aldagairen eremua zenbaki osoen multzoa dela suposatuko dugu.*

- Izan bedi $S_1 = \{(x,4)\}$ egoera, $S_1(\exists y\ \exists z\ (y*z = x \wedge y = z)) = true$ izango da, zeren eta

  $S_1(y*z = x \wedge y = z)^{2,2}_{y,z} =$ $S_1(2*2 = x \wedge 2 = 2) =$

  $(2*2 = 4 \wedge 2 = 2) = true$ bait da.
- Izan bedi $S_2 = \{(y,1), (z,2)\}$ egoera, $S_2(\forall x\ (x > 0 \rightarrow z*x > y)) = true$ izango da,

  $S_2 (x > o \rightarrow 2*x > 1)^{a}_{x} = true$ bait da edozein a zenbakirentzat (betiere osoen eremuan).
- Azkenik, izan bedi $S_3 = \{(y,2), (z,1)\}$, $S_3(\forall x(x > 0 \rightarrow z*x > y)) = false$ da,

  $S_3(x>0 \rightarrow 2*x>2) = (1 > 0 \rightarrow 2 > 2) =$ false bait da.

Gure helburua programen kontrol-puntuetan gertatzen diren egoera-multzoak formulen bidez adieraztea da, nolabait konputazio-egoeren ezaugarri komunak dira agerian uzten direnak. Testuinguru honetan erabilitako formulei asertzio deitzen zaie.

Asertzio batek egiazkoa deneko egoera-multzoa errepresentatzen du (egoera-multzo hau infinitua izan daiteke). Hortaz, T-k egoera-multzo osoa errepresentatzen du eta F-k egoera-multzo hutsa. Esate baterako, programaren puntu batean x aldagaiak zenbaki lehen bat duela esan nahi bagenu, asertzioa honako hau litzateke:

$$\text{lehena\_da}\ (x) = x \geq 2 \wedge \forall y \forall z(x = y*z \rightarrow (y=1 \ \vee \ y =x))$$

3.4. *adibidea*: *3.2 adibidean azaldutako programak asertzio hauek edukiko lituzke:*

```
begin     {x,y≥0}
   x: = a;  y : = b;      { x=a ∧ y=b ∧ x,y≥0}
   while   y > 0   do
      begin              {x+y=a+b ∧ x≥a≥0 ∧ 0<y≤b}
         x:=x+1; y:=y-1
      end
   end;    {x=a+b ∧ y=0 ∧ x,a,b≥0}
```

ϕ formula ψ baino gogorragoa dela esaten da ϕ-k errepresentatzen duen egoera-multzoa ψ-k errepresentatzen duenaren partea bada. Kasu honetan ψ ϕ baino ahulagoa dela esaten da. Formula gogorragoek murriztapen gehiago ezartzen diete beren aldagai libreei. Honen arabera erraz konpreni daiteke "true" beste edozein formula baino ahulagoa dela eta "false" dela formularik gogorrena. ϕ formula ψ baino gogorragoa baldin bada, orduan $S(\phi \rightarrow \psi)$=true izango da edozein S egoeratan.

3.5. *adibidea*: *Formula gogorrago eta ahulagoak:*

(a) $(x=5 \wedge z=25) \rightarrow z=x^2$

(b) $(x=A[i] \wedge 1 \leq i \leq n) \rightarrow \exists k(x=A[k] \wedge 1 \leq k \leq n)$

(c) $\forall k(x \leq k \leq z \rightarrow A[k] \neq p) \rightarrow A[(x+z) \underline{\text{div}}\ 2] \neq p$

(d) false $\rightarrow \phi$, edozein ϕ-rentzat

(e) $\phi \rightarrow$ true , edozein ϕ-rentzat

Eta, azkenik, programen egiaztapenerako berebizikoa den eragiketa batez arituko gara, aldibereko ordezpenaz.

Izan bitez ϕ formula, x aldagaia eta t gaia, ϕ formulan x-ren agerpen libre guztiak t gaiaz ordeztu ondoren lortutako formula honela denotatuko dugu: ϕ_x^t

Batzuetan, ϕ-n lehendik ere bazegoen aldagairen bat t gaian ageri denean, ordezpena egitean aldagai-talka gertatzen da. Aldagai-talkarik gertatzen ez denean ϕ-k x-ri buruz eta ϕ_x^t–k t-ri buruz gauza bera diote. Hau ezin daiteke baieztatu, ordea, talkarik gertatzen bada.

3.6. *adibidea*: *Izan bedi* $\phi=(\forall z\ (1<z<d \rightarrow x \underline{mod}\ z \neq 0) \wedge d<x)$ *formula eta* ϕ*-ren gainean egindako ordezpenak:*

a) $\phi_x^5 = \forall z\ (1<z<d \rightarrow 5 \underline{\text{mod}}\ z \neq 0) \wedge d<5$

b) $\phi_x^{A[i]} = \forall z\ (1<z<d \rightarrow A[i] \underline{\text{mod}}\ z \neq 0) \wedge d<A[i]$

c) $\phi_d^{d+1} = \forall z\ (1<z<d+1 \rightarrow x \underline{\text{mod}}\ z \neq 0) \wedge d+1<x$

d) $\phi_d^x = \forall z\ (1<z<x \rightarrow x \underline{\text{mod}}\ z \neq 0) \wedge x<x$

e) $\phi_x^{x \bullet z} = \forall z\ (1<z<d \rightarrow x*z \underline{\text{mod}}\ z \neq 0) \wedge d<x*z$

a), b) eta c) adibideetan ez da formulen esanahia aldatzen (gaiak bai, baina gaiei buruzkoa ez). Aldiz, d) eta e) adibideetan formulen semantika erabat aldatzen da, talken kariaz.

Era berean, $x_1, ..., x_n$ aldagaiak eta $t_1, ..., t_n$ gaiak emanez gero, x_i $(1 \leq i \leq n)$ aldagai guztiak t_i $(1 \leq i \leq n)$ gaiez <u>aldi berean</u> ordeztuta lortzen den formula $\phi_{x_1,x_2,...,x_n}^{t_1,t_2,...,t_n}$ formaz idazten da.

Aldagai-talkari buruzkoak kasu honetan ere badu zentzurik.

3.7. *adibidea*: *Izan bedi* $\phi = 1 \le i \le n \wedge A[i]=x \wedge \forall z\ (1 \le z < i \rightarrow A[z] \neq x)$ *formula:*

a) $\phi_{i,x}^{3,'a'} = 1 \le 3 \le n \wedge A[3]='a' \wedge \forall z\ (1 \le z < 3 \rightarrow A[z] \neq 'a')$

b) $\phi_{i,x}^{i+1,A[i]} = 1 \le i+1 \le n \wedge A[i+1]=A[i] \wedge \forall z\ (1 \le z < i+1 \rightarrow A[z] \neq A[i])$

a) adibidean ez dago aldagai-talkarik, baina b)-n bai.

ϕ eta ϕ_x^t formulak parekoak izango dira, baldin eta ϕ formulan x-ren agerpen libreak eta ϕ_x^t formulan t-ren agerpen libreak leku berdinetan gertatzen badira (gai baten agerpena librea da bere aldagai guztiak libreak badira).

Bi formula baliokideak dira egoera-multzo bera errepresentatzen badute, hau da, biek gauza bera baieztatzen badute.

ϕ eta ϕ_x^z parekoak badira, orduan $\exists x\phi$ eta $\exists z\phi_x^z$ eta, era berean, $\forall x\phi$ eta $\forall z\phi_x^z$ formulak elkarren baliokideak dira. t gaia f(z) erako funtzioa denean ere gauza bera gertatzen da. Hortaz, ordezpenak talkarik sortzen ez duenean, formulen kuantifikazioak baliokideak dira.

3.8. *adibidea*:

a) Izan bedi $\phi = x > 0 \wedge \exists s\ (s \ge z)$,
$\forall x\phi$ eta $\forall z(\phi_x^z) = \forall z\ (z > 0 \wedge \exists s\ (s \ge z))$ ez dira baliokideak baina
$\forall x\phi$ eta $\forall p\ (\phi_x^p) = \forall p\ (p > 0 \wedge \exists s\ (s \ge z))$ baliokideak dira.

b) Izan bedi ϕ formula, $x_1, ..., x_i$ sekuentziaren batura S dela baieztatzen duena:
$\exists i \left(S = \sum_{j=1}^{i} x_j \right)$ eta $\exists i \left(S = \sum_{j=1}^{i+1} x_j \right)$ baliokideak dira
$\exists i\ \phi$ eta $\exists i\ \phi_i^{i+1}$ erakoak dira, hurrenez hurren.

c) $\forall i\ (1 \le i \le n \rightarrow x=A[i])$ eta $\forall i\ (1 \le i+1 \le n \rightarrow x=A[i+1])$ formulak ere baliokideak dira.

Ariketak

3.1. Idatzi ondoko baieztapenak lehen mailako formulez:

- Osozko A[1..n] array-aren i-garren elementuaren ondoren dauden guztiak zeroak dira.
- A[1..n] array-an x badago, i..j sekzioan egongo da.
- A[1..n]-k bi zero dauzka.
- i posizioak A[1..n] array-a erdibitu egiten du, x baino txikiagoak batetik eta x baino handiagoak bestetik.
- f karaktere-fitxategian ez daude bi txurigune jarraian.
- A[1..n] array-ak badauka zehazki k zerotako sekzio bat.
- S[1..n] eta A[1..m] array-ek amankomunean dituzten elementuen artean X da txikiena.
- A[1..n] array-eko elementu guztiak gorantz ordenatuta daude B[1..k] array-an ($1 \leq k \leq n$), baina errepikapenik gabe.
- DES true izango da baldin eta S[1..n] array-eko elementu guztiak elkarren artean desberdinak badira eta E[1..n] array-an daudenekiko ere desberdinak badira eta false izango da bestela.
- S[1..n]-ko edozein elementu bakoiti bere ondorengo guztien baturaren berdina da.
- Ez dago B[1..n,1..n] array-an ezein elementurik positiboa izan eta ezkerraldeko behe-triangeluan ageri denik.
- S[1..n]-ko balio hertsiki positibo guztiak B[1..k] array-eko elementu bakoitien indizeak dira.
- P indizeak erakusten duen B[1..n,1..n]-ko zutabean elementu guztiak gorantz ordenatuta daude.
- f testu-fitxategiaren lerro baten hasieran kokatuta gaude.
- f testu-fitxategiaren lehen lerroan kokatuta gaude.

4. PROGRAMEN ZUZENTASUNA

4.1. TESTAK

Behin programa bat idatzi eta egon litezkeen errore sintaktiko guztiak konpondutakoan, hurrengo pausoa programak espero genituen emaitzak lortzen dituela egiaztatzea da. Prozesu hori proben bidez egin ohi da, aldez aurretik finkatutako datu-multzoez baliatuta. Baina, zoritxarrez, probak eginda ere, gertatzen dira ustegabeko emaitzak. Batzuetan errore logikoak dira emaitza okerren kausa, beste batzuetan datu "arraroek" eragiten dituzte erantzun desegokiak eta badira hardwarearen mugek sortzen dituzten ustegabekoak ere. Aukera guzti horiek eduki behar dira kontuan testak burutzean. Guk hemen, labur bada ere, banan-banan jorratuko ditugu:

1. Errore logikoak:

Errore logikoa programan egindako akatsa da. Akats hori dela medio programak ez ditu beti emaitza zuzenak bueltatuko. Errore logikodun programek emaitza zuzenak lor ditzakete zenbaitetan edo gehienetan. Baina noizbait emaitza okerretara iristea nahikoa da programari onartezin dela erizteko. Hemen ez du balio esateak: "nire programak ia-ia beti ondo funtzionatzen du". Edo beti dabil ondo, edo ez dabil.

2. Datu onartezinak:

Batzuetan programari iristen zaizkion datuak ez dira beharko luketen bezalakoak. Akatsa datuetan dago, beraz. Baina, hori horrela izanda ere, programak datu onartezin horiek identifikatzeko gai izan behako luke eta, horrezaz gain, baita tratamendu egokiak burutzeko ere.

Hara zernolako portaera eskatzen zaion programari horrelakoetan:

a) Datu onartezin bakoitzeko mezu bat azaldu beharko lioke erabiltzaileari, datuaren agerpena salatuz.
b) Ez du galeraziko gainontzeko datu zuzenen tratamendua.
c) Programa ez da gelditukо lehenengo datu okerra aurkitutakoan. Ondorenean etor litezkeen erroreak ere agerian utzi behar dira.

Lehenago ere azpimarratu den bezala, espezifikazioak finkatu behar du zein den datuen onarpen-muga eta nolako eragina izango duen programak datu onargarrietan nahiz onartezinetan.

3. Hardwarearen mugak:

Batzuetan exekuzio-erroreen zioa hardwarearen ezaugarrietan dago, hardwareak inposatzen dizkigun mugetan, alegia. Arruntak dira, besteak beste, irteerako dispositiboko lerro berean karaktere gehiegi idatzi nahi izatea, edo zenbaki handiegiekin lan egitea. Arazo hauek errore logikoekin erlazionaturik daude, azken batean hardwarearen mugak oso kontuan izan behar bait dira diseinatzean.

Testuinguru honetan programa sendoak idaztea izaten da edozein programatzaileren helburua. Programa sendoak beti bueltatuko ditu emaitza zuzenak, hori egitea posible bada, eta beti emango ditu azalpenak, emaitza zuzenak lortzea ezinezkoa bada. Datu onartezinen bat dagoelako programa sendoaren exekuzioa ez da bertan behera geldituko. Badute, gainera, beste ezaugarri garrantzitsu bat: aldaketak, egokitzapenak eta hobekuntzak programa osoa berrantolatu gabe egin daitezke. Eta hori oso ezaugarri nabarmena da, izan ere, bai bait dakigu programak maiz samar aldatzen direla bizitza-zikloan zehar.

Mantenimendu-fasean komeni da beste bi aspektu hauek ere aintzakotzat hartzea:

a) Aldaketek lehen zuzena zen programa ez-zuzen bihur dezakete. Arazo honi aurre egiteko modurik egokiena programak ondo dokumentatzea da.
b) Programaren egitura galdu egin daiteke aldaketen ondorioz. Aldaketa asko eta sakonak egin nahi badira mesedegarri gerta liteke programa berridaztea.

Aipatu ditugun erroreak ebitatzeko ez dago botika magikorik. Baina, hori bai, guztiz komenigarria da espezifikazioak idaztea (datu onargarriak eta onartezinak zein diren garbi azalduz, besteak beste), diseinua metodikoki garatzea (problema nagusia azpiproblematan banatuz, ondoz ondoko finketak eginez,...) eta erroreak atzemandakoan, adabakiak jartzea baino, egituraketari eutsiz diseinu nagusiko moduluetan aldaketak txukun burutzea.

Erroreen detekzioa

Bi dira, batez ere, erroreen detekzioan erabiltzen diren metodoak: mahaigaineko azterketa, diseinatzean egiten dena, eta programaren testa, exekutatuz gauzatzen dena. Baterako nahiz besterako beharrezkoa da proba-datuak ondo aukeratzea. Gainera, proba-datuak aukeratzean ordurarte erdi ezkutuan edo oharkabean pasatako aspektuak azaleratu egin daitezke.

Probetarako datu-multzo bat baino gehiago behar izaten da. Kasu guztiek eduki behar dute lekua datu horietan, arruntak eta orokorrak ezezik, kasu berezi edo ez-arruntak ere bai.

Mahaigaineko azterketan programaren atal desberdinen "exekuzioa" eskuz egiten da. Zenbat eta lehenago aurkitu errorea orduan eta arazo gutxiago sortuko du errore-arazketak. Horregatixe, ez da komeni programaren exekuzioen zai egotea probak egiteko.

Programaren testa programa exekutatuz burutzen da, horretarako apropos aukeratutako datuekin. Gaur egungo goi-mailako lengoaien sistemek badituzte oso lagungarri gertatzen diren tresnak. Programatzaileak tresna horiek erabiltzen jakin behar du, bai errore-detekzioan, bai arazketan.

4.2. EGIAZTAPENA

Edsger Dijkstra irakasleak esana da: "Testaren bidez akatsak badaudela erakuts daiteke, ez ordea akatsik ez dagoenik".

Hargatik, programen zuzentasuna egiaztatzeko metodo formalen beharrean gaude.

4.2.1. Semantika axiomatikoa eta frogapen formalak

Lengoaia formaletan idatzitako propietateez hausnarketarik egin nahi denean beharrezkoa da oinarri-oinarrizko egiak, zierto betetzen direnak, finkatzea eta horien gainean gero eta propietate konplexuagoak egiaztatzeko gauza izatea. Bi gai horiek jorratu eta bilduz gero arrazonatzeko kapaz den sistema eratzen da. Horrelakoei sistema formal deitu ohi zaie.

Sistema (edo kalkulu) formala, hortaz, axiomez (oinarrizko propietateez) eta inferentzi erregelez (propietate batzuetatik beste batzuk ondorioztatzeko arauez) osatzen da.

Inferentzi erregelak $\frac{P_1, P_2, \ldots, P_K}{P}$ eredukoak dira eta adierazi, P_1, P_2, ..., P_K propietateetatik P propietatea ondoriozta daitekeela adierazten dute.

Behin sistema formal bat eratuz gero, edozein P propietateren frogapena sistema formal batean burutzeak P_1, ..., $P_K = P$ sekuentzia bat aurkitzea esan nahi du, non P_i bakoitza, axioma bat edota P_1, ...,P_{i-1}-etatik inferentzi erregela batez ondoriozta daitekeen propietatea izango den.

4.1. *adibidea*: *Ondokoa programen arteko baliokidetasunaz arrazonatzeko sistema formala da, suposatuz programak asignazioen konposaketa sekuentzialez baizik ez direla osatzen.*

Axiomak

1.- x:=y ; y:=x = x:=y
2.- x:=y ; x:=z = x:=z
3.- x:=y ; z:=x = z:=y ; x:=y
4.- x:=y ; z:=y = z:=y ; x:=y

Erregelak

1.- $\dfrac{s_1 = s_2}{s_2 = s_1}$ (berdintasunaren trukakortasuna)

2.- $\dfrac{s_1 = s_2,\ s_2 = s_3}{s_1 = s_3}$ (iragankortasuna)

3.- $\dfrac{s_1 = s_2}{s;\ s_1 = s;\ s_2}$ (aurreko eransketa)

4.- $\dfrac{s_1 = s_2}{s_1;\ s = s_2;\ s}$ (atzeko eransketa)

Frogapen formalaren adibidea: c:=a; a:=c; a:=b; c:=a = a:=b; c:=b

1.-	c:=a ; a:=c = c:=a	1. Axioma
2.-	c:=a ; a:=c ; a:=b; c:=a = c:=a , a:=b ; c:=a	1 eta 4. Erregela
3.-	a:=b; c:=a = c:=b , a:=b	3. Axioma
4.-	c:=a ; a:=b ; c:=a = c:=a , c:=b ; a:=b	3 eta 3. Erregela
5.-	c:=a ; a:=c ; a:=b; c:=a = c:=a , c:=b ; a:=b	2, 4 eta 2. Erregela
6.-	c:=a ; c:=b = c:=b	2. Axioma
7.-	c:=a ; c:=b ; a:=b = c:=b ; a:=b	6 eta 4. Erregela
8.-	c:=a ; a:=c ; a:=b; c:=a = c:=b , a:=b	5, 7 eta 2. Erregela
9.-	c:=b; a:=b = a:=b ; c:=b	4. Axioma
10.-	c:=a ; a:=c ; a:=b; c:=a = a:=b ; c:=b	8, 9 eta 2. Erregela

P programa jakin bat partzialki zuzena izango da (ϕ,ψ) espezifikazioarekiko, baldin eta soilik baldin Pren exekuzioa ϕ betetzen den egoera batean hasiz gero, bukatuko dela jota, ψ betetzen den egoera batean bukatzen bada. Honela denotatuko dugu zuzentasun partziala: $\{\phi\}\ P\ \{\psi\}$.

Zuzentasun partzialaren inguruko baieztapenez arrazonamenduak eta frogapenak egin ahal izateko, sistema formal baten beharrean aurkitzen gara. Sistemaren axiomak, zuzentasun partzialari buruzko oinarrizko egiak izango dira eta inferentzi erregelak, $\{\phi\}\ P\ \{\psi\}$ erako propietate batzuetatik beste konplexuago batzuk ondorioztatzeko moduak.

Hain zuzen ere horixe da Hoare-ren sistema formala (edo axiomatikoa), programen zuzentasun partzialaz arrazonatzeko axiomen eta inferentzi erregelen multzoa.

Orohar, edozein P programatarako programazio-lengoaia bat finkatuz gero (PASCALa gure kasuan),

Axiomak: Programazio-lengoaia horretako edozein oinarrizko agindurentzat betetzen diren zuzentasun partzialari buruzko propietateak izango dira.

Inferentzi erregelak: Agindu konposatuentzako propietateak nola ondorioztatzen diren, beren baitan dituzten aginduen propietateetan oinarrituz.

Honela, programazio-lengoaiaren edozein eraikuntza sintaktikoren esanahia formalki, axiomatikoki, finkatzen da.

Hoare-ren sistema formalak programazio-lengoaiaren semantika axiomatikoa ezartzen du.

4.2.2. Hoare-ren sistema formala

Asignazioaren axioma (AA)

Asignazioaren axioma honelako hirukotea da:

$$\{\phi_x^t\} \quad x:=t \quad \{\phi\}$$

eta honela irakur daiteke:

"Asignazioa exekutatu aurretik t-k betetzen duen propietatea, x:=t exekutatu ondoren, x-k betetzen du".

4.2. *adibidea*: *Ondoko lau baieztapenak asignazioaren axiomatzat har daitezke:*

- $\{2a\ da\ lehen\ zenbaki\ lehena\}\ x:=2\ \{x\text{-}a\ da\ lehen\ zenbaki\ lehena\}$
- $\{x+y = 10\}\ x:=x+y\ \{x = 10\}$
- $\{true\}\ x:=y\ \{x = y\}$
- $\{\exists j\ (x =2^j)\}\ z:=x\ \{\exists j\ (z =2^j)\}$

Boskarren hau, ordea, ez da asignazioaren axioma:

- $\{true\}\ x:=x+y\ \{x=x+y\}$, izan ere, hau baieztatzea ondoko baldintza y=0 dela esatea bait da.

Asignazioaren semantika adierazteko darabilgun eragiketa aldibereko ozdezpena da, zeinek aldagai libreetan bakarrik bait du eragina. Asertzioek ondo definituta egon behar dute dauden kontrol-puntuko edozein konputazio-egoeratan, eta horretarako aldagai libre guztiek balio bana eduki behar dute asignatuta egoera horietan. Hortaz, programako aldagaiek asertzioetan libreak izan behar dute. Zentzugabekeria da kuantifikatuak agertzea eta nahastea besterik ez du sortzen.

$\{\exists j\ (x=2^j)\}\ j:=j+1\ \{\exists j\ (x=2^j)\}$ esate baterako, asignazioaren axioma da, baina asertzioetan agertzen den j-k ez du asignazioan agertzen denarekin zerikusirik, izena salbu.

$\{\exists k\ (x=2^k)\}\ j:=j+1\ \{\exists k\ (x=2^k)\}$ axiomak gauza bera baieztatzen du, baina askozaz argiago.

Sekuentzi konposaketaren erregela (KPE)

Bi aginduen konposaketak (ϕ,ψ) espezifikazioa bete dezan, beharrezkoa da γ tarteko asertzio bat egotea. γ asertzioa lehenengo aginduaren ondoko baldintza izango da (betiere suposatuz hasieran ϕ betetzen dela) eta bigarrengoaren aurreko baldintza, hau da, γ tarteko asertzioak errepresentatzen duen edozein egoeratan bigarrengo agindua exekutatuz, ψ betetzen den egoera lortzen da.

Honenbestez, zera esan dezakegu:

$$\frac{\{\phi\}\ I_1\ \{\gamma\}\ ,\ \{\gamma\}\ I_2\ \{\psi\}}{\{\phi\}\ \underline{\text{begin}}\ I_1\ ;\ I_2\ \underline{\text{end}}\ \{\psi\}}$$

non γ tarteko asertzioa den.

Eta, orokorrago:

$$\frac{\{\phi_1\}\ I_1\ \{\phi_2\}\ ,\ \{\phi_2\}\ I_2\ \{\phi_3\}\ ,\ \ldots\ ,\ \{\phi_n\}\ I_n\ \{\phi_{n+1}\}}{\{\phi_1\}\ \underline{\text{begin}}\ I_1\ ;\ I_2\ ;\ \ldots;\ I_n\ \underline{\text{end}}\ \{\phi_{n+1}\}}$$

ϕ_2 , ... , ϕ_n : tarteko asertzioak.

4.3. ***adibidea***: *Egiazta ezazu bi aldagaien balioa elkartrukatzen duen ondorengo programa:*

$\{x=a \wedge y=b\}$ <u>begin</u> lag:=x; x:=y; y:=lag <u>end</u> $\{x=b \wedge y=a\}$

1.- $\{x=a \wedge y=b\}$ lag:=x $\{lag=a \wedge y=b\}$ (AA)
2.- $\{lag=a \wedge y=b\}$ x:=y $\{lag=a \wedge x=b\}$ (AA)
3.- $\{lag=a \wedge x=b\}$ y:=lag $\{y=a \wedge x=b\}$ (AA)
4.- $\{x=a \wedge y=b\}$ <u>begin</u> lag:=x; x:=y; y:=lag <u>end</u> $\{y=a \wedge x=b\}$ 1,2,3 eta (KPE)

Ondorioaren erregela (ODE)

Pentsa dezagun zergatik ote den egiazkoa ondokoa:

$\{x \geq y\}$ x:=x-y $\{x \geq 0\}$

Arrazoi sinple batengatik: $x \geq y \rightarrow x-y \geq 0$ betetzen delako eta, gainera, $\{x-y \geq 0\}\ x:=x-y\ \{x \geq 0\}$ Asignazioaren Axioma delako.

Ondorengoetan ere beste hainbeste gertatzen da:

1.- $\{1 \leq i < n\}\ i:=i+1\ \{1 \leq i \leq n\}$ egiazkoa da,
batetik $(1 \leq i < n \rightarrow 1 \leq i+1 \leq n)$ eta,
bestetik $\{1 \leq i+1 \leq n\}\ i:=i+1\ \{1 \leq i \leq n\}$ AAa da eta.

2.- $\{1 \leq k \leq n\}\ x:=A[k]\ \{\exists i\ (1 \leq i \leq n \wedge x=A[i])\}$ ere egiazkoa da, izan ere,
$\{1 \leq k \leq n\}\ x:=A[k]\ \{1 \leq k \leq n \wedge x=A[k]\}$ AAa da eta
$1 \leq k \leq n \wedge x=A[k] \rightarrow \exists i\ (1 \leq i \leq n \wedge x=A[i])$

3.- $\{\forall i\ (1<i<d \rightarrow x \underline{\text{mod}}\ i \neq 0) \wedge x \underline{\text{mod}}\ d \neq 0\}\ d:=d+1\ \{\forall i\ (1<i<d \rightarrow x \underline{\text{mod}}\ i \neq 0)\}$ egiazkoa da:
$(\forall i\ (1<i<d \rightarrow x \underline{\text{mod}}\ i \neq 0) \wedge x \underline{\text{mod}}\ d \neq 0) \rightarrow \forall i\ (1<i<d+1 \rightarrow x \underline{\text{mod}}\ i \neq 0)$ eta
$\{\forall i\ (1<i<d+1 \rightarrow x \underline{\text{mod}}\ i \neq 0)\}\ d:=d+1\ \{\forall i\ (1<i<d \rightarrow x \text{ mod } i \neq 0)\}$ AAa da.

Orain arte egindako gogoeten formalizazioa ondorioaren erregela da:

$$\frac{\varphi_1 \rightarrow \varphi_2\ ,\ \{\varphi_2\}\ P\ \{\psi_1\}\ ,\ \psi_1 \rightarrow \psi_2}{\{\varphi_1\}\ P\ \{\psi_2\}}$$

Kasu partikular gisa ikus daitezke $\varphi_1 = \varphi_2$ eta $\psi_1 = \psi_2$ direnekoak.

4.4. ***adibidea*****:** *Frogatu ondokoa:*

$$\{i<n \wedge s=\sum_{k=1}^{i} A[k]\}$$

<u>begin</u>

i:=i+1;

s:=s+A[i]

<u>end</u>

$$\{i\leq n \wedge s=\sum_{k=1}^{i} A[k]\}$$

Frogapena:

1.- $(i<n \wedge s=\sum_{k=1}^{i} A[k]) \rightarrow (i+1\leq n \wedge s+A[i+1]=\sum_{k=1}^{i+1} A[k])$

2.- $\{i+1\leq n \wedge s+A[i+1]=\sum_{k=1}^{i+1} A[k]\}$

$i:=i+1$

$\{i\leq n \wedge s+A[i]=\sum_{k=1}^{i} A[k]\}$ (AA)

3.- $\{i<n \wedge s=\sum_{k=1}^{i} A[k]\}\ i:=i+1\ \{i\leq n \wedge s+A[i]=\sum_{k=1}^{i} A[k]\}$ 1, 2 eta (ODE)

4.- $\{i\leq n \wedge s+A[i]=\sum_{k=1}^{i} A[k]\}$

$s:=s+A[i]$

$\{i\leq n \wedge s=\sum_{k=1}^{i} A[k]\}$ (AA)

5.- $\{i<n \wedge s=\sum_{k=1}^{i} A[k]\}$

$i:=i+1;\ s:=s+A[i]$

$\{i\leq n \wedge s=\sum_{k=1}^{i} A[k]\}$ 3, 4 eta (KPE)

4.5. *adibidea*: *Frogatu programa honek bi aldagaien balioak elkartrukatzen dituela:*

begin x:=x+y ; y:=x-y ; x:=x-y end

1.-	$(x = a \wedge y = b) \rightarrow (x+y = a+b \wedge y = b)$	
2.-	$\{x+y = a+b \wedge y = b\}$ x:=x+y $\{x = a+b \wedge y = b\}$	(AA)
3.-	$\{x = a \wedge y = b\}$ x:=x+y $\{x = a+b \wedge y = b\}$	1, 2, (ODE)
4.-	$(x = a+b \wedge y = b) \rightarrow (x = a+b \wedge x-y = a)$	
5.-	$\{x = a+b \wedge x-y = a\}$ x:=x-y $\{x = a+b \wedge y = a\}$	(AA)
6.-	$\{x = a+b \wedge y = b\}$ x:=x-y $\{x = a+b \wedge y = a\}$	4, 5, (ODE)
7.-	$(x = a+b \wedge y = b) \rightarrow (x-y = b \wedge y = a)$	
8.-	$\{x-y = b \wedge y = a\}$ x:=x-y $\{x = b \wedge y = a\}$	(AA)
9.-	$\{x = a+b \wedge y = a\}$ x:=x-y $\{x = b \wedge y = a\}$	7, 8, (ODE)
10.-	$\{x = a \wedge y = b\}$ x:=x+y ; y:=x-y; x:=x-y $\{x = b \wedge y = a\}$	3, 6, 9, (KPE)

Baldintzazko konposaketaren erregelak (BDE)

Pascal-ez baldintzazko agindu bat baino gehiago badago eta bakoitzari dagokion inferentzi erregelaz arituko gara.

If-then-else ereduko aginduen espezifikazio batekiko zuzentasun partziala frogatzeko zera ziurtatu behar dugu, aurreko baldintza betetzen den edozein egoeratatik, bai konputazioak then-aren bidea hartzen badu, bai else-rena hartzen badu, azkenean ondoko baldintza betetzen den egoera batera ailegatzen dela. Hau da:

$$\frac{\{\phi \wedge B\}\ I_1\ \{\psi\}\ ,\ \{\phi \wedge \neg B\}\ I_2\ \{\psi\}}{\{\phi\}\ \underline{\text{if}}\ B\ \underline{\text{then}}\ I_1\ \underline{\text{else}}\ I_2\ \{\psi\}}$$

4.6. *adibidea*: *Froga ezazu ondoko programak z aldagaian x eta y-ren arteko kenduraren balio absolutua uzten duela:*

$\{true\}$ if $x \geq y$ then $z:=x-y$ else $z:=y-x$ $\{z = |x-y|\}$

1.- $x \geq y \rightarrow x-y \geq 0$
2.- $\{x-y \geq 0\}\ z:=x-y\ \{x-y \geq 0 \wedge z=x-y\}$ (AA)
3.- $(x-y \geq 0 \wedge z=x-y) \rightarrow (z = |x-y|)$
4.- $\{x \geq y\}\ z:=x-y\ \{z = |x-y|\}$ 1, 2, 3, (ODE)
5.- $\{\neg\, x \geq y\}\ z:=y-x\ \{z = y-x \wedge \neg\, x \geq y\}$ (AA)
6.- $(z = y-x \wedge \neg\, x \geq y) \rightarrow (z = |x-y|)$
7.- $\{\neg\, x \geq y\}\ z:=y-x\ \{z = |x-y|\}$ 5, 6, (ODE)
8.- $\{true\}$ if $x \geq y$ then $z:=x-y$ else $z:=y-x$ $\{z = |x-y|\}$ 4, 7, (BDE)

Eta zer gertatzen ote da else-rik ez badago?

$\{\phi\}$ if B then I $\{\psi\}$ frogatu behar badugu ere bi kasuak hartu behar ditugu kontuan. Izan ere, B bete ala ez, ψ betetzen den egoera batean bukatzen dela ziurtatu behar bait dugu. Beraz, $\neg B$ betez gero $\phi \wedge \neg B$ egiazkoa deneko egoera batean egongo gara eta, inolako agindurik egikaritzen ez denez, ψ ere egiazkoa izango da egoera horretan. Dagokion inferentzi erregela, esandakoak esanda, hauxe:

$$\frac{\{\phi \wedge B\}\ I\ \{\psi\}\ ,\quad \phi \wedge \neg B \rightarrow \psi}{\{\phi\}\ \text{if}\ B\ \text{then}\ I\ \{\psi\}}$$

4.7. *adibidea*: Frogatu x eta y-ren arteko maximoa uzten dela x-n eta minimoa y-n:

$\{x = a \wedge y = b\}$
if $x < y$ then begin lag:=x; x:=y; y:=lag end
$\{x = \max(a,b) \wedge y = \min(a,b)\}$

Frogapena:

1.- $(x = a \wedge y = b \wedge x < y) \rightarrow (y = \max(a,b) \wedge x = \min(a,b))$
2.- $\{y = \max(a,b) \wedge x = \min(a,b)\}$
begin lag:=x; x:=y; y:=lag end
$\{x = \max(a,b) \wedge y = \min(a,b)\}$ (lehen frogatutakoa 4.3 adibidean)
3.- $\{x = a \wedge y = b \wedge x < y\}$
begin lag:=x ; x:=y ; y:=lag end
$\{x = \max(a,b) \wedge y = \min(a,b)\}$ 1,2, (ODE)
4.- $(x = a \wedge y = b \wedge x \geq y) \rightarrow (x = \max(a,b) \wedge y = \min(a,b))$
5.- $\{x = a \wedge y = b\}$
if $x < y$ then begin lag:=x ; x:=y ; y:=lag end
$\{x = \max(a,b) \wedge y = \min(a,b)\}$ 3, 4, (BDE)

4.8. *adibidea*: *Programa-zati honek A[1..n] osozko array-aren baliorik handiena kalkulatzen du m aldagaian. Egiazta ezazu:*

$$\underline{\text{begin}}\ \{1 \le i < n \ \wedge\ \exists j\,(1 \le j \le i \ \wedge\ m = A[j]) \ \wedge\ \forall j\,(1 \le j \le i \ \rightarrow\ m \ge A[j])\}$$
$$\quad i:=i+1;$$
$$\quad \underline{\text{if}}\ m < A[i]\ \underline{\text{then}}\ m:=A[i]$$
$$\underline{\text{end}}\ \{1 \le i \le n \ \wedge\ \exists j\,(1 \le j \le i \ \wedge\ m = A[j]) \ \wedge\ \forall j\,(1 \le j \le i \ \rightarrow\ m \ge A[j])\}$$

Frogapena:

1.- $(1 \le i < n \wedge \exists j\,(1 \le j \le i \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i \rightarrow m \ge A[j])\} \rightarrow$
$(1 \le i+1 \le n \wedge \exists j\,(1 \le j \le (i+1)-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le (i+1)-1 \rightarrow m \ge A[j]))$

2.- $\{1 \le i+1 \le n \wedge \exists j\,(1 \le j \le (i+1)-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le (i+1)-1 \rightarrow m \ge A[j])\}$
$i:=i+1$
$\{1 \le i \le n \wedge \exists j\,(1 \le j \le i-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i-1 \rightarrow m \ge A[j])\}$ (AA)

3.- $(1 \le i \le n \wedge \exists j\,(1 \le j \le i-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i-1 \rightarrow m \ge A[j]) \wedge m < A[i]) \rightarrow$
$(1 \le i \le n \wedge \forall j\,(1 \le j \le i-1 \rightarrow A[i] > A[j]))$

4.- $\{1 \le i \le n \wedge \forall j\,(1 \le j \le i-1 \rightarrow A[i] > A[j])\}$
$m:=A[i]$
$\{1 \le i \le n \wedge \forall j\,(1 \le j \le i-1 \rightarrow m > A[j]) \wedge m = A[i]\}$ (AA)

5.- $(1 \le i \le n \wedge \forall j\,(1 \le j \le i-1 \rightarrow m > A[j]) \wedge m = A[i]) \rightarrow$
$(1 \le i \le n \wedge \forall j\,(1 \le j \le i \rightarrow m \ge A[j]) \wedge \exists j\,(1 \le j \le i \wedge m = A[j]))$

6.- $\{\ 1 \le i \le n \wedge \exists j\,(1 \le j \le i-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i-1 \rightarrow m \ge A[j]) \wedge m < A[i]\ \}$
$m:=A[i]$
$\{1 \le i \le n \wedge \exists j\,(1 \le j \le i \wedge m > A[j]) \wedge \forall j\,(1 \le j \le i \rightarrow m \ge A[j])\}$ 3, 4, 5 eta (ODE)

7.- $(1 \le i \le n \wedge \exists j\,(1 \le j \le i-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i-1 \rightarrow m \ge A[j]) \wedge m \ge A[i]) \rightarrow$
$(1 \le i \le n \wedge \exists j\,(1 \le j \le i \wedge \forall j\,(1 \le j \le i \rightarrow m \ge A[j])\ )$

8.- $\{\ 1 \le i \le n \wedge \exists j\,(1 \le j \le i-1 \wedge m = A[j]) \wedge \forall j\,(1 \le j \le i-1 \rightarrow m \ge A[j]) \wedge m < A[i]\ \}$
$\underline{\text{if}}\ m < A[i]\ \underline{\text{then}}\ m:=A[i]$
$\{1 \le i \le n \wedge \exists j\,(1 \le j \le i \wedge m > A[j]) \wedge \forall j\,(1 \le j \le i \rightarrow m \ge A[j])\}$ 6, 7 et (BDE)

9.- $\underline{\text{begin}}\ \{1 \le i < n \ \wedge\ \exists j\,(1 \le j \le i \ \wedge\ m = A[j]) \ \wedge\ \forall j\,(1 \le j \le i \ \rightarrow\ m \ge A[j])\}$
$\quad i:=i+1;$
$\quad \underline{\text{if}}\ m < A[i]\ \underline{\text{then}}\ m:=A[i]$
$\underline{\text{end}}\ \{1 \le i \le n \ \wedge\ \exists j\,(1 \le j \le i \ \wedge\ m = A[j]) \ \wedge\ \forall j\,(1 \le j \le i \ \rightarrow\ m \ge A[j])\}$
1, 2, (ODE), 8 eta (KPE)

Kasu-hautaketa agindua (KHE)

Hauxe dugu frogagaia:

$\{\phi\}$ case e of
$s_1 : I_1;$
…
$s_k : I_k$
end
$\{\psi\}$

Agindu honen egikaritzapena, horretara, ϕ betetzen den egoera batetik abiatu eta exekutatzen den edozein I_i-rentzat ψ betetzen den egoera batean bukatzen da. Egiaztapenean $\{\phi \wedge e = s_i\}\ I_i\ \{\psi\}$ erako baieztapenak frogatu behar dira, edozein izanda ere 1 eta k-ren artean hartzen den i-ren balioa ($1 \leq i \leq k$).

K baieztapenok frogagarriak izan arren, gerta liteke I_i-rik ez egikaritzea, e aldagaia ezein s_i etiketaren berdina ez izateagatik. Kasu honetan exekuzio-errore baten aurrean aurkitzen gara. Lehen aipatutako k baieztapenek soilik baldin hasierako egoerak ($e = s_1$ v...v $e = s_k$) betetzen badu bermatuko dute nahi genuena. Inferentzi erregela honako hau litzateke:

$$\frac{\{\phi \wedge e = s_1\}\ I_1\ \{\psi\},\ \ldots,\ \{\phi \wedge e = s_k\}\ I_k\ \{\psi\}}{\{\phi \wedge (e = s_1\ v \ldots v\ e = s_k)\}\ \underline{\text{case}}\ e\ \underline{\text{of}}\ s_1 : I_1;\ \ldots;\ s_k : I_k\ \underline{\text{end}}\ \{\psi\}}$$

Case aginduak ***Otherwise*** klausula badauka inferentzi erregela nabarmen aldatzen da:

$\{\phi\}$ case e of
$s_1 : I_1;$
…
$s_k : I_k$
otherwise I_{k+1}
$\{\psi\}$

($e = s_1$ v...v $e = s_k$) betetzen ez denean, edo ($e \neq s_1 \wedge \ldots \wedge e \neq s_k$) betetzen denean, I_{k+1} egikaritu behar da eta bide honetatik ere azkenean ψ betetzen dela bermatu behar da.

Beraz:

$$\frac{\{\phi \wedge e=s_1\}\ I_1\ \{\psi\}\ ,\ldots,\ \{\phi \wedge e=s_k\}\ I_k\ \{\psi\}\ ,\ \{\phi \wedge e\neq s_1 \wedge \ldots \wedge e\neq s_k\}\ I_{k+1}\ \{\psi\}}{\{\phi\}\ \underline{\text{case}}\ e\ \underline{\text{of}}\ s_1: I_1;\ \ldots;\ s_k: I_k\ \underline{\text{otherwise}}\ I_{k+1}\ \{\psi\}}$$

Erazagupenak

Hoare-ren sistema formalean $\{\phi\}P\{\psi\}$ hirukotearen frogapena $P_1, P_2,..., P_n$ sekuentzia finitua da, non $P_n = \{\phi\}\ P\ \{\psi\}$ eta $P_i\ (1 \leq i \leq n)$ bakoitza ondorengo aukeretatik bat den:

a) Axioma baten instantzia den Hoare-ren hirukotea
b) $P_1, P_2,..., P_{i-1}$ hauetatik inferentzi erregela batez ondoriozta daitekeen Hoare-ren hirukotea
c) Lehen mailako formula

Koka gaitezen c) kasuan. Zein dira zuzentasun partzialaren frogapenean idazten ditugun lehen mailako formulak? Orain artean $(\phi \rightarrow \psi)$ erako formulak erabili ditugu ondorioaren erregela aplikatzeko. Esate baterako:

1) $x > 0 \rightarrow 2*x \geq 2$
2) $x > 0 \rightarrow x \geq 1$
3) $x > 0 \wedge y > 0 \rightarrow x*y \neq 0$
4) $x = a \wedge x \geq 0 \rightarrow x = |a|$
5) $\neg\ \text{odd}\ (x) \rightarrow \text{odd}\ (x+1)$
6) $(\forall j\ (1 \leq j \leq i\text{-}1 \rightarrow m \geq A[j]) \wedge m < A[i]) \rightarrow \forall j\ (1 \leq j \leq i \rightarrow A[i] \geq A[j])$

Azken finean, datu-mota jakin bateko objektu eta eragiketen propietateak baieztatzen dituzten formulak dira. Honela, 1) eta 2) egiazkoak dira x osoa bada, ez ordea erreala bada. 3) biderkaketaren propietate baten ondorioa da, hala osoetan nola errealetan. 4) balio absolutuaren propietateaz lortzen da, 5) zenbaki bikoitien propietateen ondorio baizik ez da. Eta, azkenik, 6) $\leq$ erlazioaren iragankortasunaren fruitu da. Lehen mailako formula guzti hauek, erabiltzen dituzten objektu-moten propietateengatik betetzen dira.

Ondorioz, programa baten erazagupen-atalak eragin zuzena du egiaztapenean, frogapen formalean erabiliko diren propietateak finkatzen bait ditu.

Pascal-eko programen erazagupen-atalean hiru azpiatal bereiziko ditugu: konstanteak, aldagaiak eta motak.

- Konstante-definizio bakoitzak erazagututako berdintza axioma gisa ezartzen du (n=20, adibidez) gure sistema formalean.
- *"var x:T"* aldagai-erazagupenak, x-k T motako aldagaia izateagatik betetzen dituen propietate guztiak ezartzen ditu axiomak bezala.
- *"type TN=T"* mota-erazagupenaren ondorioz TNk Tren propietate guztiak hartzen ditu. Era honetan, TN motako aldagai guztiek propietate hauek bereganatzen dituzte.

  Laburtuz, har dezagun *D; I* programa, non

  D: erazagupen-atala den,

  I: agindu-multzoa den,

  eta Dk ezarritako axiomen multzoari H deituko diogun;

Hortaz:

$$\frac{H \vdash \{\phi\}\ I\ \{\psi\}}{\{\phi\}\ D;\ I\ \{\psi\}}$$

M |– P notazioaz zera adierazten dugu: "P propietatea frogagarria da M multzoko axiomak erabiliz". Nahiz eta erregelan H multzo osoa adierazten den, praktikan programaren zuzentasunean eragiten duten Hko propietateak baino ez dira erabiltzen.

Ariketak

4.1. Frogatu formalki, 4.1. adibideko sistema formalean oinarrituta:

a:=b ; d:=a ; b:=a ; d:=a = d:=b ; a:=b

4.2. Egiaztatu ondoko Hoare-ren hirukotea:

{x=a ∧ y=b} begin lag:=x; x:=y; y:=lag end {x=a ∧ y=b ∧ lag=a}

4.3. A[1..n] osozko array-a, eta j eta b aldagai osoak direla, frogatu:

$$\text{begin} \quad \left\{1 \le j < n \wedge b = \sum_{i=1}^{j-1} A[i]\right\}$$

b := b + A[j];

j:= j + 1

$$\text{end} \quad \left\{1 \le j \le n \wedge b = \sum_{i=1}^{j-1} A[i]\right\}$$

4.4. Frogatu zuzentasun partzialari buruzko ondorengo baieztapenak, kontuan hartuz s_i-k Fibonacciren segidaren i-garren gaia adierazten duela:

(a) begin { $1 \le i < n \wedge x = s_i \wedge y = s_{i+1} \wedge z = s_{i+2}$ }

x := y;

y := z;

z := x+y;

i := i+1

end { $1 \le i \le n \wedge x = s_i \wedge y = s_{i+1} \wedge z = s_{i+2}$ }

(b) begin { $\exists i(i \ge 0 \wedge u = s_i \wedge z = s_{i+1})$ }

u := u+z;

z := u+z

end { $\exists i(i \ge 0 \wedge u = s_i \wedge z = s_{i+1})$ }

4.5. A[1..n] eta B[1..n] osozko array-ak, eta k eta r aldagai osoak direla, frogatu:

```
begin  { b=true ∧ ∀i(1≤i≤k → B[i]=A[i]*r) }
     k := k+1;
     b:= (B[k] <> A[k]*r )
end     { b=true ↔ ∀i(1≤i≤k → B[i]=A[i]*r) }
```

4.6. Erabaki baieztapen hau egiazkoa ala faltsua den eta erantzuna arrazoitu.

```
{x=a}
    if odd(x) then y:=x+1
{x=a  ∧ not (odd (y))}
```

4.7. Asmatu inferentzi erregela egoki bat ondoko baldintzazko aginduarentzat:

```
case
     B1 then I1;
     B2 then I2;
          …
     Bk then Ik
end
```

non B_1, B_2, …, B_k baldintza boolearrak diren (ez derrigorrez elkar-bartertzaileak). Agindu honen semantika I_i exekutatzea da, i hori B_j egiazkoa deneko j txikiena ($1 \leq j \leq k$) dela, eta ez badago betetzen den baldintzarik inongo j-rentzat ez du ezer egiten.

4.8. 4.7. ariketako baldintzazko aginduari otherwise klausula erantsi, dagokion aldaketa semantikoa modurik naturalenean hartuz. Lortu ondoren inferentzi erregela bat agindu berriarentzat.

4.9. Izan bedi 'adi' adirazpena, eta izan bitez e_1, e_2, e_3 eta e_4 adi-ren mota bereko balio desberdinak. Ondoko aginduak adi ebaluatu eta bere balioa edozein e_i-ren (non i=1,2,3,4) berdina bada I_i, I_{i+1}, … sententziak sekuentzialki exekutatzen ditu, harik eta end edo break agindu bat topatu arte. Adi-ren ebaluaketak ez badu e_i baliorik ematen, agindua amaitu egiten da ezertxo ere egin gabe.

```
case adi of
    e1: I1;
    e2 : I2;
break
    e3 : I3;
    e4 : I4;
end
```

Asmatu agindu honentzako inferentzi erregela bat.

Iterazioak eta inbariantearen kontzeptua

{ϕ} *while* *B* *do* *I* *{ψ}* frogatu nahi izanez gero kontuan hartu behar da, jeneralean, I behin baino gehiagotan exekutatzen dela konputazio berean eta, ondorioz, B baldintzaren balioztapenaren aurretik dagoen kontrol-puntuari dagokion asertzioak ere egoera bat baino gehiago errepresentatu behar duela. Iterazioa zentzuzkoa bada, egoera guzti hauek propietate komunen bat eduki behar dute. Esate baterako:

```
begin
        i:=0; s:=0;
        INB=   {A-ren lehenengo i elementuen batura s da
               eta 0≤i≤n}
        while i<>n do begin
                    i:=i+1;
                    s:=s+A[i];   } I
               end
end
```

Propietate hau, I-k (iterazioaren barruko agindu-multzoak) maneiatzen dituen objektuena da eta kontserbatu egiten du. Hau da, I-k objektuak aldatu, aldatzen ditu, baina balio berriek delako propietatea mantentzen dute. Hortik datorkio inbariante izena. Hala ere, garbi utzi behar da inbariantea ez dela I-k kontserbatzen duen edozein propietate, baizik eta bigiztaren semantika ezartzen duena, hain juxtu. Inbarianteak iterazio-puntuko egoera posibleen multzoa karakterizatzen du eta, hortaz, baita iterazioaren eragina ere.

Aurreko adibidean ($0 \le i \le n$) ere iterazioak kontserbatzen duen propietatea da, baina iterazioaren eraginari buruz (s aldagaiak pauso bakoitzean A array-eko landu den zatiaren batura du) ez dio ezer nabarmenik.

Orokorrean, x=x, "true" edo edozein tautologia I-k (edozein hartuta ere) kontserbatzen du, baina ez dituzte inolaz ere iterazio-puntuko konputazio-egoerak errepresentatzen, baizik eta egoera-multzo osoa.

Aurreko adibidera itzuliz, jo dezagun $INB = (s = \sum_{j=1}^{i} A[j] \ \wedge \ 0 \le i \le n)$ propietatea inbariantea dela, hau da, bigiztaren gorputzak kontserbatzen duela. Alegia:

{INB ∧ i≠n} begin i:=i+1; s:=s+A[i] end {INB}

Gainera, hasierako egoeran (i=0 ∧ s=0) dugunez, bereziki inbariantea iterazioaren hasierako egoeran beteko da:

(i=0 ∧ s=0) → INB

Beraz, inbariantea hasieran egiazkoa dela eta urrats bakoitzean mantentzen dela jakinda, zer edo zer baiezta al dezakegu iterazioa bukatzen deneko egoeraz? Bada, kontserbazioz, inbariantea egiazkoa izango da eta, gainera, bukatu dugunez, B baldintza boolearrak false balioa hartu du.

Gure kasuan, ondokoa betetzen da:

$$s = \sum_{j=1}^{i} A[j] \ \wedge \ 0 \le i \le n \ \wedge \ \neg (i \neq n),$$

horretara:

$$(s = \sum_{j=1}^{i} A[j] \ \wedge \ i = n) \quad \text{edo, beste modu batez,} \quad s = \sum_{j=1}^{n} A[j] \ .$$

Beraz, programak zierto betetzen du espezifikazioa (zuzentasun partziala).

Inbariantetzat (0 ≤ i ≤ n) hartu izan balitz, kontserbatzen dela eta hasieran betetzen dela froga genezakeen, baina ondorioztatuko genukeena ondokoa litzateke: Iterazioa bukatzen denean (0 ≤ i ≤ n ∧ ¬ i ≠ n) betetzen da, edo 0 ≤ i ≤ n nahi izanez gero.

0 ≤ i ≤ n formulak ez du ezer asko adierazten iterazioari buruz, ez behintzat benetan egiten duena ondorioztatzeko adina. Hots, 0 ≤ i ≤ n, iterazioak kontserbatzen duen propietatea da, baina ez du bere semantika adierazten: ez da iterazioaren inbariantea.

While agindua (WHE)

Bada, {ϕ} while B do I {ψ} erako zuzentasun partzialeko propietateak frogatzeko INB propietate inbariante bat aurkitu behar dugu, I-k kontserbatuko duena eta iterazioaren semantika adieraziko diguna. Eta inbariantea finkatu ondoren hiru ezaugarriok frogatu beharko ditugu:

. I-k benetan kontserbatzen duela, hau da, {INB ∧ B} I {INB}

. Hasieran betetzen dela: ϕ → INB

. Iterazioa bukatutakoan ondoko baldintza beteko dela: INB ∧ ¬B → ψ

Honenbestez, inferentzi erregela hau izango da:

$$\frac{\phi \rightarrow INB,\ \{INB \wedge B\}\, I\, \{INB\},\ INB \wedge \neg B \rightarrow \psi}{\{\phi\}\ \underline{\text{while}}\ B\ \underline{\text{do}}\ I\ \{\psi\}}$$

4.9. *adibidea*: *Array bateko elementurik handiena aurkitzen duen programaren zuzentasun-egiaztapena:*

```
begin {n ≥ 1}
        i:=1; m:=A[1];
        while i<>n do
                begin
                        i:=i+1;
                        if m < A[i] then m:=A[i]
                end
end   {∀j (1 ≤ j ≤ n → m ≥ A[j]) ∧ ∃j (1 ≤ j ≤ n ∧ m = A[j])}
```

Inbariantea hauxe izango da:

INB = {1 ≤ i ≤ n ∧ ∀j (1 ≤ j ≤ i → m ≥ A[j]) ∧ ∃j (1 ≤ j ≤ i ∧ m = A[j])

eta frogatutzat joko dugu:

1.- {INB ∧ i≠n} begin i:=i+1; if m < A[i] then m:=A[i] end {INB}

Frogapena osatzeko:

2.- {n≥1} i:=1 {n≥1 ∧ i =1} (AA)
3.- {n≥1 ∧ i =1} m:=A[1] {n≥1 ∧ i=1 ∧ m = A[1]} (AA)
4.- (n≥1 ∧ i=1 ∧ m = A[1]) →
(n≥1 ∧ ∀j (1 ≤ j ≤ 1 → A[1] ≥ A[j]) ∧ ∃j (1 ≤ j ≤ 1 ∧ A[1] = A[j])) →
INB
5.- (INB ∧ ¬(i≠n)) → (∀j (1 ≤ j ≤ n → m ≥ A[j]) ∧ ∃j (1 ≤ j ≤ n ∧ m = A[j]))
6.- {i=1 ∧ m=A [1]}
while i<>n do begin i:=i+1; if m<A[i] then m:=A[i] end
{∀j (1≤ j ≤ n → m ≥ A[j]) ∧ ∃j (1≤ j ≤ n ∧ m=A[j])} 1, 4, 5, (WHE)
7- {n≥1}
begin i:=1; m:=A[1]; while i<>n do begin i:=i+1; if m<A[i] then m:=A[i] end end
{∀j (1≤ j ≤ n → m ≥ A[j]) ∧ ∃j (1≤ j ≤ n ∧ m=A[j])} 2,3,6 (KPE)

Arestian ikusitako WHEren formulazioa sinplifika daiteke, frogapenetan ondorioaren erregela ere erabiliz. Honela geratuko litzaiguke WHE sinplifikatua:

$$\frac{\{INB \wedge B\}\ I\ \{INB\}}{\{INB\}\ \underline{while}\ B\ \underline{do}\ I\ \{INB \wedge \neg B\}}$$

Repeat agindua (RPE)

Aurrekoaren antzera, repeat iterazioak ere bere eragina deskribatuko duen asertzio inbariante bat behar du. Simetriaz kontsidera dezagun repeat-aren asertzio inbariantea B baldintzaren aurreko kontrol-puntuari dagokiola. Hau honela izanik, repeat aginduaren erregelak forma hau hartzen du:

$$\frac{\{\phi\}\ I\ \{INB\},\ \{INB \wedge \neg B\}\ I\ \{INB\},\ INB \wedge B \rightarrow \psi}{\{\phi\}\ \underline{repeat}\ I\ \underline{until}\ B\ \{\psi\}}$$

Froga daiteke erregela hau zuzena dela WHE erregelaz eta BALKID ondoko baliokidetzaz baliatuz: repeat I until B = begin I; while ¬B do I end.

Frogapena honakoa litzateke:

1.- $\{\phi\}\ I\ \{INB\}$
2.- $\{INB \wedge \neg B\}\ I\ \{INB\}$
3.- $INB \wedge B \rightarrow \psi$
4.- $\{INB\}$ while $\neg B$ do $I\ \{INB \wedge \neg\neg B\}$ — 2 eta (WHE)
5.- $(INB \wedge \neg\neg B) \rightarrow (INB \wedge B)$
6.- $\{\phi\}$ begin I; while $\neg B$ do $I\ \{INB \wedge B\}$ — 1,4,5,(ODE) eta (KPE)
7.- $\{\phi\}$ begin I; while $\neg B$ do $I\ \{\psi\}$ — 6,3 eta (ODE)
8.- $\{\phi\}$ repeat I until B $\{\psi\}$ — 7 eta BALKID

Erregela honek agerian uzten du I agindua gutxienez behin egikaritzen dela. Honek while-ren kasuan egin genuen sinplifikazioa eragozten du. Kasu honetan erregela sinplifikatua honela geratzen da:

$$\frac{\{\phi\}\ I\ \{INB\},\ \{INB \wedge \neg B\}\ I\ \{INB\}}{\{\phi\}\ \underline{repeat}\ I\ \underline{until}\ B\ \{INB \wedge B\}}$$

4.10 ***adibidea*****:** *1^2, 2^2, 3^2, ... segidaren lehenengo n gaien batura kalkulatzen duen ondoko programaren zuzentasun-frogapena:*

begin:

$\{n \geq 1\}$

$s := 0;\ x := 0;$

$\{n \geq 1 \wedge s = 0 \wedge x = 0\}$

repeat

$x := x + 1\ ;\ s := s + x^2$

$$\text{INB} = \left\{ s = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right\}$$

until $x = n$

end;

$$\left\{ s = \sum_{i=1}^{n} i^2 \right\}$$

1.- $\{n \geq 1\} s := 0;\ x := 0\ \{n \geq 1 \wedge s = 0 \wedge x = 0\}$ (A.A.)

2.- $\{n \geq 1 \wedge s = 0 \wedge x = 0\}\ x := x + 1\ ;\ s := s + x^2\ \{n \geq 1 \wedge s = 1 \wedge x = 1\}$

3.- $(n \geq 1 \wedge s = 1 \wedge x = 1) \rightarrow \left(s = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right)$

4.- $\{n \geq 1 \wedge s = 0 \wedge x = 0\}\ x := x + 1\ ;\ s := s + x^2\ \left\{ s = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right\}$ 2, 3, (ODE)

5.- $\left(\text{INB} \wedge \neg(x = n)\right) \rightarrow \left(s = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x < n \right) \rightarrow \left(s + (x+1)^2 = \sum_{i=1}^{x+1} i^2 \ \wedge\ 1 \leq x < n \right)$

6.- $\left\{ s + (x+1)^2 = \sum_{i=1}^{x+1} i^2 \ \wedge\ 1 \leq x < n \right\}\ x := x + 1\ \left\{ s + x^2 = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right\}$ (A.A.)

7.- $\left\{ s + x^2 = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right\}\ s := s + x^2\ \left\{ s = \sum_{i=1}^{x} i^2 \ \wedge\ 1 \leq x \leq n \right\}$ (A.A.)

8.- $\left\{ s = \sum_{i=1}^{x} i^2 \ \wedge \ 1 \le x < n \right\}\ x := x+1 \ ;\ s := s + x^2 \ \left\{ s = \sum_{i=1}^{x} i^2 \ \wedge \ 1 \le x \le n \right\}$

5, 6, 7 (ODE) eta (KPE)

9.- $\left(\text{INB} \ \wedge \ (x = n)\right) \rightarrow \left(s = \sum_{i=1}^{x} i^2 \wedge 1 \le x \le n \ \wedge x = n \right) \rightarrow \left(s = \sum_{i=1}^{n} i^2 \right)$

10.- $\{ n \ge 1 \ \wedge \ s = 0 \ \wedge \ x = 0 \}$

repeat $x := x+1 \ ;\ s := s + x^2$ until $x = n$

$\left\{ s = \sum_{i=1}^{n} i^2 \right\}$ 4, 8, 9, (RPE)

11.– $\{ n \ge 1 \}$ begin … end $\left\{ s = \sum_{i=1}^{n} i^2 \right\}$ 1, 10, (KPE), (ODE)

Programen bukaeraren arazoa

Dagoeneko aztertu dugu nola froga daitekeen iterazioak dituzten programen zuzentasun partziala. Baina ϕ aurreko baldintza eta ψ ondoko baldintzaz emandako espezifikazioarekiko P iteraziozko programaren zuzentasun partziala frogatzean, azken finean, zera diogu:

$\{\phi\}$ P $\{\psi\}$ = ϕ betetzen den egoera batetik abiatuz, Pren edozein konputazio, bukatzen bada, ψ betetzen den egoera batean bukatzen da.

Orain, programak partzialki ezezik osoki zuzenak izan daitezen nahi dugu.

Alegia,

$\{\phi\}$ [P] $\{\psi\}$ = ϕ betetzen den egoera batean hasten den Pren edozein konputazio ψ betetzen den egoera batean bukatzen da.

Pk iterazioriK edo errekurtsibitaterik ez duenean $\{\phi\}$ P $\{\psi\}$ = $\{\phi\}$ [P] $\{\psi\}$ betetzen da. Baina iterazioren bat badu, zuzentasun partzialaz gain BUK(ϕ,P) frogatu beharko da, hau da, ϕ betetzen den egoera batean hasten den Pren edozein konputazio urrats-kopuru finituan bukatzen dela.

Horretara, $\{\phi\}\,[P]\,\{\psi\} = \{\phi\}\,P\,\{\psi\} + \text{BUK}(\phi, P)$

4.11. *adibidea*: *Izan bedi i zenbaki osoa eta har dezagun:*

{true} <u>*while*</u> *i<>0* <u>*do*</u> *i:=i-1 {i=0}* Hoare-ren hirukotea.

Dudarik gabe baieztapen hau egiazkoa da, eta horretaz jabetzeko nahikoa da inbariantetzat true formula hartzea (iterazio honek ez du ezer egiten). Batetik, bistakoa da, gelditzen bada, $i \neq 0$ baldintza boolearra faltsua (i=0) egiten delako gelditzen dela. "True" aurreko baldintzarekiko, ordea, ez da egiazkoa ondoko baieztapena:

{true} [<u>*while*</u> *i<>0* <u>*do*</u> *i:=i-1] {i=0}*

Ez da egia, ezen i-ren balio negatiboentzat ez bait da betetzen.

Aitzitik, *{i ≥ 0} [* <u>*while*</u> *i<>0* <u>*do*</u> *i:=i-1] {i=0}* egiazkoa da.

Beraz, bukaeraren arazoa aurreko baldintza batekiko definitu behar da. Horrexegatik, hain zuzen, erabiltzen da ϕ argumentua BUK predikatuan.

4.12. *adibidea*: *Izan bitez, x,y,z,r osoak, eta ondorengo bi programak:*

```
P programa                    K programa
begin                         begin
 r:=0; z:=x;                   r:=0; z:=x;
 while z<>y do                 while z<y do
  begin                        begin
        z:=z+1;                       z:=z+1;
        r:=r+1                        r:=r+1
  end                          end
end;                          end;
```

Bi programa hauek ez dira $`\{true\}\ [\ ]\ \{r = y - x\}'$ espezifikoarekiko osoki zuzenak, baina arrazoi desberdinengatik:

1. Pk bukaera-arazoa du espezifikazio horrekiko,
2. Kk zuzentasun partzialaren inguruko arazo bat du.

Ondorioz, ez dira zuzenak baieztapen hauek,

$\{true\}\ [P]\ \{r = y - x\}$

$\{true\}\ [K]\ \{r = y - x\}$

1.- {true} P {r=y-x} zuzena da, baina BUK (true,P) ez, eta

2.- {true} K {r=y-x} ez da zuzena, nahiz eta BUK (true, K) baden.

Zer gertatu behar du *while B do I* edo *repeat I until B* iterazioak buka daitezen, hau da, urrats-kopuru finitua eman dezaten?

Gutxienez Ik Bn ageri diren aldagaietako bat aldatu beharko du. Beharrezkoa da baldintza hau, eta 4.11 adibideko programak betetzen du: while i<>0 do i:=i-1. Halere, beharrezkoa bai, baina ez da nahikoa baldintza hau betetzea: 4.11 adibideko programa ez da bukatzen "true" aurreko baldintzarekiko, nahiz eta i≥0 baldintzarekiko amaitzen den. Beraz, zein betebehar ezarri behar zaio aldaketa horri bukaera ziurta dezagun?

Aski da aurreko baldintzak errepresentatzen dituen egoeretan aldaketa hori soilik aldi-kopuru finituan aplikatu ahal izatea B faltsu bihurtu gabe. 4.11 adibidean, i:=i-1 asignazioa, i>0 denean, soilik aldi-kopuru finituan aplika daiteke i≠0 faltsu egin gabe,eta i<0 denean, berriz, i ≠ 0 kontserbatuz infinituki aplika daiteke.

4.12. adibideko K programan z:=z+1 aldaketa, z=x dela, soilik aldi-kopuru finituan exekuta daiteke z<y faltsu bilakatu arte; aldiz, P programan z=x deneko egoera batetik abiatuz aldi-kopuru infinituan exekuta daiteke agindu hori z≠y faltsu bihurtu gabe.

Laburtuz, eta orain artean esandakoaren arabera, bi baldintza ezar daitezke programen bukaeraz ziurtasuna edukitzeko:

- iterazioaren gorputzak "zerbait" (aldagaiak, orain arteko adibideetan) aldatu behar du,
- aurreko baldintza betetzen den egoera batetik abiatuta, delako "zerbait" hori soilik aldi-kopuru finituan alda daiteke iterazioaren baldintza faltsu bihurtu gabe.

Goazen, bada, ikuspegi intuikor hau multzoetara egokitzera:

```
{ϕ}   while B do
      begin    {INB ∧ B}
      …                  {b0, b1, ..., bn, ...} multzoa hartuko dugu
      …                  kontuan, non bi Ik i-garren iterazioan
      …                  "aldatzen duenaren" balioa bait da.
      end;               Multzo honek finitua izan behar du.
```

Ik aldatzen duenak hartzen dituen balioen multzoa formalizatu eta iterazio bat noiz bukatzen den frogatzeko "ondo oinarritutako ordena" kontzeptua definituko dugu ondoren.

Iterazioen bukaera eta ondo oinarritutako ordenak

Izan bitez S multzo ez-hutsa eta $\sqsubseteq$ erlazio bitarra, (S, $\sqsubseteq$) ondo oinarritutako ordena izango da baldin eta soilik baldin,

a) $\subseteq$ erlazioa Sn ordena partziala bada. Hots, ondoko propietateak betetzen baditu:

- Bihurkorra:

$$s \subseteq s \quad \forall s \in S$$

- Antisimetrikoa:

$$(s \subseteq r \ \wedge \ r \subseteq s) \rightarrow (s = r) \quad \forall s, r \in S$$

- Iragankorra:

$$(s \subseteq r \ \wedge \ r \subseteq z \rightarrow s \subseteq z) \quad \forall s, r, z \in S$$

b) Sn ez dago $\subseteq$-rekiko beheranzko katea infiniturik. Hots, ez dago inolako $< x_i \in S / i \in \mathbb{N} >$ sekuentziarik non $x_0 \supset x_1 \supset \ldots \supset x_k \supset \ldots$.

$\supset$ ikurraren bidez beheranzko ordena hertsia adierazten da:

$$x \supset y \leftrightarrow (y \subseteq x \ \wedge \ y \neq x)$$

4.13. *adibideak*:

1.- $(.^{\cdot}, \leq)$ ordena partziala da eta ez dago beheranzko katea infiniturik. Badago, ordea, goranzko katea infiniturik.

$(, \leq)$ eta $(, \leq)$ ez dira ondo oinarritutako ordenak, bai bait daude beheranzko katea infinituak.

2.- $(\wp^f(M), \subseteq)$, non $\wp^f(M)$ Mko azpimultzo finitu guztien multzoa bait da. Erraz froga daiteke ordena partziala dela. Eman dezagun badagoela $M_1 \supset M_2 \supset \ldots M_k \supset \ldots$ beheranzko katea infinitu bat, non halabeharrez M_1, M_2, ..., M_k, ..., Mren azpimultzo finituak izango diren. Beraz, M_1 infinitu azpimultzo dauzkan multzo finitua izango da, eta hau absurdua da nabarmenki.

Gauza bera esan liteke $(\wp(M), \subseteq)$ bikoterako, non M multzo finitua den.

3.- $(.^{\cdot 2}, \prec)$, non $(x,y) \prec (u,v)$ b.s.b. $x<u$, edo $x=u$ eta $y \leq v$. Ordena partziala da, bistan denez. Jo dezagun badela beheranzko katea infinitu bat:

$$KAT = (x_0, y_0) \succ (x_1, y_1) \succ \ldots \succ (x_k, y_k) \succ (x_{k+1}, y_{k+1}) \ldots,$$ non edozein k-rentzat $x_k > x_{k+1}$ edota $(x_k = x_{k+1}$ eta $y_k > y_{k+1})$. Honela bada, KAT AZPIkateez osaturik egongo da:

$$AZPI(x) = (x, z_0) \succ (x, z_1) \succ \cdots (x, z_r) \succ \cdots, \text{ non } z_0 > z_1 > \ldots > z_r > \ldots$$

eta gisa honetan:

$$KAT = AZPI\,(x_0)\ AZPI\,(x_1) \ldots AZPI(x_m) \text{ non } x_0 > x_1 > \ldots > x_m > \ldots$$

Beraz, KAT infinitua da soilik baldin azpikatea infiniturik badago edo azpikate-kopurua infinitua bada. Bata nahiz bestea ezinezkoak dira $(\mathbb{N}, \leq)$ ondo oinarritutako ordena bait da: ondorioz katea ezin daiteke infinitua izan.

4.- $\left(\{a\}^*, \subseteq\right)$ non $s \subseteq t$ b.s.b. $luz(s) \leq luz(t)$. Nabaria da ordena partziala dela. Sekuentziak luzera finituzkoak direnez ez dago katea infiniturik luzerarekiko beheranzko ordenan.

While agindurako metodoa

Izan bedi *{INB}* *while* *B* *do* *I* iterazioa; iterazio hau INB inbariantearekiko bukatzen dela frogatzeko ondo oinarritutako ordena bat (S, $\subseteq$) eta e adierazpen bat (Ik aldatu eta Bn ageri diren aldagaiez osatua) aurkitu behar dira eta ondokoa frogatu:

a) e adierazpenak ondo oinarritutako ordenako balioak hartzen ditu:

$$INB \wedge B \rightarrow e \in S$$

eta

b) iterazioro e beheratu egiten da ordena horrekiko:

$$\{INB \wedge B \wedge e = b\}\ I\ \{b \supset e\}$$

Metodo hau justifikatuta geratzen da $b_0, b_1, \ldots, b_n, \ldots$ balio-zerrendan oinarrituta, non b_i horietako bakoitza e-k Iren i-garren exekuzioaren ondoren hartzen duen balioa den. Hortaz:

a) $b_0, b_1, \ldots, b_n, \ldots \in S$

eta

b) $b_0 \supset b_1 \supset \ldots \supset b_n \supset \ldots$

$(s, \subseteq)$ ondo oinarritutako ordena denez katea hau finitua da.

4.14. *adibidea*: *Ondorengo programak bi dimentsioko array baten elementu guztiak lantzen ditu. Programa honen bukaera aztertuko dugu:*

```
var A: array [1..n,1..m] of T;
begin {n≥1 ∧ m≥1}
i:=1; j:=1;
          INB={P(A,i,j) ∧ 1≤i≤n+1 ∧ 1≤j≤m}
while i ≤ n and j ≤ m do
 begin {P(A,i,j) ∧ 1≤i≤n ∧ 1≤j≤m}
  landu (A[i,j]);
  if j<m      then j:=j+1
      else    begin
                i:=i+1;
                j:=1
              end
 end
end {P (A, n+1, m) }
```

Ondo oinarritutako ordena bat baino gehiago erabiliz froga daiteke programa bukatzen dela:

1) $(\mathbb{N}^2,\leq)$ ondo oinarritutako ordena eta $\mathbf{e}$=(n-i, m-j) adierazpena hartuz:
   a) $INB \wedge B \rightarrow 1\leq i\leq n \wedge 1\leq j\leq m \rightarrow n-i\geq 0 \wedge m-j\geq 0 \rightarrow (n-i,\ m-j)\in\mathbb{N}^2$
   b) $\{INB \wedge B \wedge (n-i,m-j)=(x_0,y_0)\}$
      if j<m then j:=j+1 else begin i:=i+1; j:=1 end;
      $\{(n-i,m-j)=(x_0,y_0-1) \vee (n-i,m-j)=(x_0-1,m-1)\}\rightarrow$
      $\{(n-i,m-j)<(x_0,y_0)\}$

2) $(\wp(M),\subseteq)$ ondo oinarritutako ordena hartuz, non
   $M=\{(a,b)/1\leq a\leq n \wedge 1\leq b\leq m\}$ bait da, eta
   $\mathbf{e}$ adierazpena = $\{(a,b)/(a=i \wedge j\leq b\leq m) \vee (i+1\leq a\leq n \wedge 1\leq b\leq m)\}$, hau da,
   landu gabeko array-zatiaren posizioen multzoa adierazten duena.

Nolanahi ere, badago problemaren ikuspegi intuikorragoa: pauso bakoitzean gutxitzen dena landu gabeko elementuen kopurua dena kontuan hartuz, eta kopuru hau

zenbaki naturala denaz jabetuz. Hari honetik, $(.\cdot,\leq)$ ondo oinarritutako ordena har dezakegu eta e=m-j+1+m∗(n-i) adierazpena, i, j, n eta m-ren arabera landu gabeko elementuen kopurua adierazten duena. Honela, bukaeraren frogapena ezezik iterazioak eman beharreko pauso-kopuruaren adierazpena ere lor daiteke.

Bukaera frogatuko dugu:

a) $INB \wedge B \rightarrow (1\leq i\leq m \wedge 1\leq j\leq m) \rightarrow m-j+1+m*(n-i) \in .\cdot$

b)

$$\{INB \wedge B \wedge m-j+1+m*(n-i) = b\} \rightarrow$$
$$\{1\leq j\leq m \wedge m-j+1+m.(n-i) = b\} \rightarrow$$
$$\{(1\leq j<m \vee 1\leq j=m) \wedge m-j+1+m*(n-i) = b\}$$
$$\underline{\text{if}}\ j<m\ \underline{\text{then}}\ j:=j+1\ \underline{\text{else}}\ \underline{\text{begin}}\ i:=i+1\ ;\ j:=1\ \underline{\text{end}};$$
$$\{(m-j+1+m*(n-i)=b-1) \vee (m-j+1+m*(n-i)=b-m+(m-1)=b-1)\} \rightarrow$$
$$\{m-j+1+m*(n-i)<b\}$$

Metodoaren egokitzapena $(.\cdot,\leq)$ ***ordenarako***

{INB} *<u>while</u> B <u>do</u> I* iterazioa emanda, bukatzen dela frogatzeko nahikoa da e adierazpen osoa (Ik aldatu eta Bn ageri diren aldagaiez osatua) aurkitzea eta ondokoa frogatzea:

a) e adierazpneak $.\cdot$-ko balioak hartzen ditu:

$$INB \wedge B \rightarrow e \in \mathbb{N}$$

b) iterazioro beheratu egiten da ≤ ordenarekiko

$$\{INB \wedge B \wedge e = b\}\ I\ \{e < b\}$$

Metodo hau justifikatuta dago, $(.\cdot,\leq)$ ondo oinarritutako ordena bait da eta, azken batean, emandako frogapen-urratsak metodo orokorraren partikularizazioa besterik ez bait dira.

Borne-adierazpen deritzo e adierazpenari, egoeraren araberako iterazio-pausoen kopurua mugatu edo bornatu egiten du eta. Aurreko adibidean pauso-kopurua adierazten zuen, izan ere pausoero bat gutxitu eta zero bihurtuz bukatzen bait zen:

$$i = 1 \wedge j = 1 \rightarrow e = m*n$$
$$i = n+1 \wedge j = 1 \rightarrow e = 0$$

4.11. adibideko programan $e=i$ adierazpenak eta 4.12 adibideko K programan $e=y-z$ adierazpenak, iterazioek konputazio-egoeraren arabera zenbat pauso ematen duten adierazten dute.

Arrazonamendu berdinak erabiliz repeat agindurako ere egokitu daiteke metodoa:

$\{INB \wedge \neg B\}$
I
$\{INB\}$
until B

Repeat agindurako metodoa

repeat I {INB} until B iterazioa $INB \wedge \neg B$ betetzen den edozein egoeratan hasi eta pauso-kopuru finituan bukatzen dela frogatzeko aski da $(S,\sqsubseteq)$ ondo oinarritutako ordena eta e adierazpena aurkitzea, eta ondokoa frogatzea:

a) $INB \wedge \neg B \rightarrow e \in S$
b) $\{INB \wedge \neg B \wedge e = b\}$ I $\{e \subset b\}$

Hemen ere, askozaz errazago eta intuikorragoa da $(.\dot{\cdot},\leq)$ ondo oinarritutako ordena erabiliz borne-adierazpenez baliatzea.

Beste edozein iteraziozko agindurentzat ere aurki daiteke a eta b propietateen ezaugarriak finkatuz bukaeraren frogapena bideratuko duen metodoren bat.

4.15. ***adibidea*****:** *4.9 eta 4.10 adibidetako programen bukaera-frogapena ikusiko dugu, $(.\dot{\cdot},\leq)$ metodoa erabiliz. Iterazio hauen zuzentasun partziala frogatu da dagoeneko.*

4.9 adibideko iterazioaren inbariantea:

$INB = \{1 \leq i \leq n \wedge \forall j(1 \leq j \leq i \rightarrow m \geq A[j]) \wedge \exists j(1 \leq j \leq i \wedge m = A[j])\}$

e=m-i, borne-adierazpena hartuta:

a) $(INB \wedge i \neq n) \rightarrow i < n \rightarrow n - 1 > 0$
b) $(INB \wedge i \neq n \wedge n - i = b) \rightarrow (n - (i+1) < b)$

```
{n-(i+1) < b}
        i:=i+1
{n-i < b}
        if m < A [i] then  m:= A [i]
{n-i < b}
```

4.10. adibideko iteraziorako inbariantea:

$$INB = \left\{ s = \sum_{i=1}^{x} i^2 \wedge 1 \leq x \leq n \right\}$$

e=n-x, adierazpena hartuz,

a) $(\text{INB} \wedge \neg(x = n)) \rightarrow x < n \rightarrow n - x > 0$

b) $(\text{INB} \wedge \neg(x = n) \wedge n - x = b) \rightarrow (n - (x + 1) < b)$

$\{n-(x+1) < b\}$

$x := x+1$

$\{n-x < b\}$

$s := s + x^2$

$\{n-x < b\}$

Bi kasuotan borne-adierazpenak pauso-kopurua finkatzen du eta, garbiro ikus daitekeenez, aginduaren zenbait zatik ez du borne-adierazpenaren balioan eraginik.

Iterazioen ez-bukaera

Aurreko baldintza batekiko programa bat bukatzen dela frogatzerik ez dagoenean, bi kasu bereiztea komeni da:

1.kasua: Aurreko baldintzak errepresentatzen duen edozein egoerarentzat ez da bukatzen. Hau baieztatzeko nahikoa da bukaera-baldintza kontserbatzen dela frogatzea.

Hau da,

while B do I, ez da gelditzen baldin $\{\text{INB} \wedge \text{B}\}$ I $\{\text{INB} \wedge \text{B}\}$

repeat I until B, ez da gelditzen baldin $\{\text{INB} \wedge \neg\text{B}\}$ I $\{\text{INB} \wedge \neg\text{B}\}$

Ohartzekoa da kasu honetan ondoko baldintza, $\text{B} \wedge \neg\text{B}$ beteko denez, faltsuaren baliokidea dela.

Zer esan nahi du ondoko baldintza faltsu izateak?

Bada, $\{\phi\}$ P {false} baieztapenak esan nahi du, P programa ez dela bukatzen ϕ betetzen den edozein egoeratan hasita.

Hortaz: {true} P {false} baieztapenak, edozein hasierako egoera hartuta ere, P ez dela sekula bukatuko esan nahi du.

4.16. *adibidea*: *Azter dezagun ondoko programaren bukaera:*

```
begin
    i:=1;
    while i≥1 do
        begin
            s:=s+A[i];
            i:=i+1
        end
end
```

Edozein hasierako egoera emanda programa hau bukatzen ez dela ikusteko, nahikoa da inbariantea ondokoa dela frogatzea:

$$INB = \left\{ s = \sum_{j=1}^{i-1} A[j] \ \wedge \ i \geq 1 \right\}$$

Dakigunez, $INB \wedge \neg B \rightarrow \psi$ gertatzen da, eta

$$INB \ \wedge \ \neg B = \left\{ s = \sum_{j=1}^{i-1} A[j] \ \wedge \ i \geq 1 \ \wedge \ i < 1 \right\}$$

Hemendik berehalakoan ondoriozta daiteke:

$$INB \wedge \neg B \rightarrow \text{false}$$

2. kasua: Aurreko baldintzak errepresentatzen dituen egoera batzuetarako ez da bukatzen, baina beste batzuetarako bai.

Aurreko baldintzak errepresentatzen duen egoera-multzoaren banaketa burutu behar da: batetik, amaitzen deneko egoera-multzoa eta, bestetik, amaitzen ez denekoa. Honetarako, programa geldi dadin aurreko baldintzari erantsi beharreko baldintza aurkitzea eta, ondoren, aurreko baldintza berritu horrekiko bukatzen dela frogatzea nahikoa da. Bestalde, aipatu baldintzaren ukapenak bukatzen ez dela frogatzeko aukera ematen du 1. kasuan ikusi dugunez.

4.17. ***adibidea*****:** *Programa honen bukaera-azterketa:*

begin $\{n \geq 1\}$

i:=1;

$$INB = \{\forall k\ (1 \leq k < i \rightarrow A[k] \neq x) \wedge\ 1 \leq i \leq n\}$$

while x<>A[i] do

if i<n then i:=i+1

end $\{\exists j(1 \leq j \leq n\ \wedge\ x = A[j]\ \wedge\ \forall k(1 \leq k < j \rightarrow A[k] \neq x))\}$

Sarrerako baldintzei $\exists j(1 \leq j \leq n\ \wedge\ x = A[j])$ murriztapena erantsiz, suposatuko dugu k aldagaiak ondoko baldintza betetzen duela:

$$x = A[k]\ \wedge\ 1 \leq k \leq n\ \wedge\ \forall r(1 \leq r < k \rightarrow x \neq A[r])$$

e=k-i adierazpena hartuz gero, erraz froga daiteke aurreko baldintza horrekiko programa gelditzen dela.

Gainera, $\neg\exists j(1 \leq j \leq n\ \wedge\ x = A[j])$ ukapena erantsiz gero, zera froga daiteke, iterazioko baldintza ($x \neq A[i]$) inbariantekoa dela.

Ariketak

4.10. Frogatu formalki:

```
begin   {n ≥ 1}
        [
            i:=0;
            s:=0;
            while i<>n do
                                begin
                                   i:=i+1;
                                   s:=s+A[i]
                                end
        ]
        {s = Σ_{j=1}^{n} A[j]}
end
```

4.11. Frogatu ondoko programak b-n x zenbakia lehena den ala ez erabakitzen duela:

```
begin
        d:=2;
        b:= true;
        while (d<>x and b) do
                begin
                        if x mod d=0 then b:= false;
                        d:=d+1
                end
end
```

4.12. Asmatu inferentzi erregela bat ondoko iteraziozko aginduarentzat:

```
do
    B1: I1;
    B2: I2;
od;
```

non, aginduaren semantika honakoa bait da:

Iteratu I_1 exekutatuz B_1 betetzen denean eta I_2 exekutatuz B_1 bete ez baina B_2 betetzen denean. Iterazioa ez B_1, ez B_2 betetzen ez denean bukatuko da. B_1 eta B_2 baldintzei ez zaie elkar-baztertzaileak izatea edo bestelako murriztapenik ezartzen.

4.13. Ondoko programa-zatiak d-n |x-y| adierazpenaren balioa uzten du. Dokumentatu tarteko asertzioez eta inbarianteaz.

```
{ }
d:=0;
if x≤y then
     begin u:=x; z:=y end
 else
     begin u:=y; z:=x end;
{ }
while u<>z do  INB={ }
     begin { }
        z:=z-1;
        d:=d+1
     end;
{d= |x-y|}
```

4.14. Ondoko programak x elementuaren bilaketa bitarra burutzen du A[1..n] osozko array ordenatuan eta dagoen posizioa 'pos' aldagaian gordetzen du (x elementua An ez badago pos-ek 0 balioa hartzen du).
Espezifika eta dokumenta ezazu:

```
begin
   l:=1; u:=n; aurkitu:=false;
   while l≤u and not aurkitu do
      begin
         pos:=(l+u) div 2;
         if A[pos]=x then aurkitu:=true
             else if A[pos]<x then l:=pos+1
                else u:=pos-1
      end
   if not aurkitu then pos:=0
end
```

4.15. A[1..n] balio osozko array-a emanda, ondoko programak Ako balio bakoitiei dagozkien indizeak lortzen ditu Bn modu ordenatuan eta Ako balio bikoitiei dagozkienak, berriz, Dn. Bukaeran Ako indize guztiak Bn edo Dn ageri dira. Idatzi programan aipatutako ϕ, INB eta ψ asertzioak.

```
var  t, z, k: integer;
begin          φ = {
    t:=1; z:=0; k:=0;
    while t<=n do   INB = {
        begin
            if odd (A [t]) then
                begin
                    z:=z+1;
                    B[z]:=t
                end
            else
                begin
                    k:=k+1;
                    D [k]:=t
                end;
            t:=t+1
        end
end;          ψ = { }
```

Array-en osagaiei egindako asignazioa (AOA)

Oker arrunta da array-a aldagai sinplez osatutako aldagai egituratua dela suposatzea. Ez, ez da horrela. Array-a balio egituratuak har ditzakeen aldagai sinplea da. Hori garbi asko ikusten da A[i]:=t erako asignazioen egiaztapenean:

$$\left\{P^{t}_{A[i]}\right\}\ A[i]:=t\ \{P\}$$

Baieztapen hau ez da erabat zuzena, A[i] t balioa hartzen duen aldagai sinpletzat hartzen bait dugu.

$A[A[i]]:=t$ bezalako asignazioetan, edo zeharkakoetan, esate baterako *begin ...; i:=A[j]; ...; A[i]:=t; ... end* daukagunean, ez dugu suposatu behar aldagai sinple bati balio bat asignatzen zaionik, aldagai sinple hori array-aren beste osagaiekiko independentea balitz bezala. Adibidez, baieztapen hau,

$$\{\text{true}\}\ \ A[A[2]]:=1\ \{A[A[2]]=1\}$$

ez da zuzena. Horretaz jabetzeko aski da hasierako egoeratzat A=(2,2) hartzea, eta konturatzea nola bukaeran lortzen dugun A=(2,1) egoerak ez duen ondoko baldintza betetzen.

Honela bada, array-a bere osotasunean hartu behar dugu eta A[i]:=t erako asignazioak A:=(A,t/i) notazioaz adieraziko ditugu, non:

$$(A,t/i)[j] = \begin{cases} t & \text{baldin } j=i \\ A[j] & \text{baldin } j\neq i \end{cases}$$

eta lor daitekeen axioma:

$$\left\{\phi_{A}^{(A,t/i)}\right\}\ A[i]:=t\ \{\phi\}$$

Honen arabera, $\{\text{true}\}\ \ A[A[2]]:=1\ \{A[A[2]]=1\}$ ez da axioma bat, ezta eratorgarria ere, eta hori frogatzeko nahikoa da ondoko ordezkapena egitea,

$$\left(A[A[2]]=1\right)_{A}^{(A,1/A[2])} = A[2]\neq 2\ \vee\ \left(A[2]=2\ \wedge\ A[1]=1\ \right),$$

eta honela lortutako aurreko baldintza *true* baino gogorragoa dela ikustea.

Ikus dezagun orain nola erabil daitekeen axioma berria frogapen batean:

4.18. adibidea: *Frogatu:*

$$\left\{\forall j\left(1\leq j<i\rightarrow A[j]=2^{j}\right)\right\}\ A[i]:=2{**}i\ \left\{\forall j\left(1\leq j\leq i\rightarrow A[j]=2^{j}\right)\right\}$$

1.– $\left(\forall j\left(1 \le j < i \rightarrow A[j] = 2^j\right)\right) \rightarrow \left(\forall j\left(1 \le j \le i \rightarrow \left(A, 2^i / i\right)[j] = 2^j\right)\right)$

2.– $\left\{\forall j\left(1 \le j \le i \rightarrow \left(A, 2^i / i\right)[j] = 2^j\right)\right\}\ A[i] := 2 ** i\ \left\{\forall j\left(1 \le j \le i \rightarrow A[j] = 2^j\right)\right\}$

Orain artean, gauzak ez gehiegi endregatzeagatik, ez dugu kontuan hartu A array-ari egindako edozein asignaziok eraginik eduki dezan bete beharreko beste baldintza bat hau dela: indizeak mugen artean egotea.

Betebehar berri hau kontuan hartuta, $A[n_1..n_2]$ array-aren osagaiei egindako asignazioaren axioma honakoa izango da:

$$\left\{n_1 \le i \le n_2 \ \wedge\ \phi_A^{(A,t/i)}\right\}\ A[i] := t\ \{\phi\}$$

For agindua (FORE)

{true} *for i:=1 to 10 do A[i]:=0* $\left\{\forall j\left(1 \le j \le 10 \rightarrow A[j] = 0\right)\right\}$ zuzentasun partzialari buruzko baieztapena frogatzeko, hamar baieztapen kateatu frogatu beharko ditugu:

$$\{\text{true}\} = \left\{\forall j\left(1 \le j < 1 \rightarrow A[j] = 0\right)\right\}$$

$A[i] := 0$

$$\left\{A[i] = 0\right\} = \left\{\forall j\left(1 \le j \le 1 \rightarrow A[j] = 0\right)\right\} = \left\{\forall j\left(1 \le j < 2 \rightarrow A[j] = 0\right)\right\}$$

$A[i] := 0$

$$\left\{A[1] = 0 \ \wedge\ A[2] = 0\right\} = \left\{\forall j\left(1 \le j \le 2 \rightarrow A[j] = 0\right)\right\} = \left\{\forall j\left(1 \le j < 3 \rightarrow A[j] = 0\right)\right\}$$

.

$A[i] := 0$

.

.

$$\left\{\forall j\left(1 \le j \le 9 \rightarrow A[j] = 0\right)\right\} = \left\{\forall j\left(1 \le j < 10 \rightarrow A[j] = 0\right)\right\}$$

$A[i] := 0$

$$\left\{\forall j\left(1 \le j \le 10 \rightarrow A[j] = 0\right)\right\}$$

Frogatu nahi duguna zuzentasun partzialari buruzko baieztapen bakar batez adieraz dezakegu:

$$\left\{1 \le j \le 10 \ \wedge\ \forall j\left(1 \le j < i \rightarrow A[j] = 0\right)\right\}\ A[i] := 0\ \left\{\forall j\left(1 \le j \le i \rightarrow A[j] = 0\right)\right\},$$

eta i desberdinentzat errepikatzen den propietatea honela jarriz gero:

$$P(i)=\left\{\forall j\left(1\le j\le i\rightarrow A[j]=0\right)\right\}$$

erraz antzeman daiteke zein den frogatu beharreko baieztapena:

$$\left\{1\le j\le 10\ \wedge\ P(\text{aurre }(i))\right\}\ \ I\ \ \left\{P(i)\right\}$$

Orokorrean, *for i:=n_1 to n_2 do I* agindua eta *begin i:=n_1; while i≤n_2 do I; i:=i+1 end* iteraziozko agindua semantikoki baliokideak dira. Azken iterazio honetan inbariantea dena, for-aren kasuan i-ren araberako propietate (P(i)) gisa adierazten da, i-ren aldaketak inplizituki gertatzen bait dira for-ean. Beraz, P(i)-k for aginduaren semantika adierazten du inbarianteak ikusi ditugun gainerako iteraziozko aginduena adierazten zuen bezalaxe.

Adibidean ikusitakotik ondoko inferentzi erregela atera daiteke:

$$\frac{\{n_1\le i\le n_2\wedge P(\text{aurre }(i))\}\ \ I\ \{P(i)\}}{\{n_1\le n_2\wedge P(\text{aurre }(n_1))\}\ \ \underline{\text{for}}\ i:=n_1\ \underline{\text{to}}\ n_2\ \underline{\text{do}}\ I\ \ \{P(n_2)\}}$$

$n_1\le n_2$ baldintza ezarriko ez bagenu $\left\{P\left(\text{aurre }(n_1)\right)\right\}$ for i:=n_1 to n_2 do I $\left\{P(n_2)\right\}$ ondoriozta daitekeela suposatuko genuke inolako $n_1>n_2$ frogapenik egin gabe.

Eta $n_1>n_2$ denean for agindua ez da exekutatzen. Kasu honetan for-aren aurreko baldintza mantendu egiten da.

Hona **for ez-exekutatuaren axioma (FEEA)**:

$$\{n_1>n_2\ \wedge\ \phi\}\ \underline{\text{for}}\ i:=n_1\ \underline{\text{to}}\ n_2\ \underline{\text{do}}\ I\ \{\phi\}$$

Suposatzen ari gara FORE-n i aldagaia eta n_1, n_2 adierazpenak ez direla aldatzen, bestela inferentzi erregela ez litzateke zuzena izango. Kontuan hartu, I-ren exekuzio-kopuru zehatz baten konposaketa egitetik abiatu garela, hasieran n_1-ek eta n_2-k dituzten balioen arteko i bakoitzarentzat konposaketa bana. n_1, n_2 edo i aldatuz gero for aginduaren semantika ez dator bat premisan suposatu dugunarekin.

FORE erregelan *aurre (n_1)* erabiltzen da eta, jeneralean, aurrekari honek ez luke zertan egon beharrik. Arazo tekniko hau ekiditeko erregela alda daitekeen arren, horretan murgiltzea premiagabeko konplikazioetan sartzea litzateke.

For baten egiaztapenerako erregela bat eta axioma bat erabiliz, eta biak konbinatuz zuzentasun partzialari buruzko baieztapen bat lortzen dugunez, guzti hau formalizatzeko **disjuntzioaren erregela (DJE)** laguntzailea definituko dugu:

$$\frac{\{\phi_1\}\ I\ \{\psi_1\},\ \{\phi_2\}\ I\ \{\psi_2\}}{\{\phi_1 \vee \phi_2\}\ I\ \{\psi_1 \vee \psi_2\}}$$

4.19. *adibidea*: *Froga ezazu ondorengo programak u eta t-ren arteko biderkadura kalkulatzen duela:*

```
begin    φ={ t ≥ 0 }
  s:=0
  for  i:=1 to t do  s:=s+u
end      ψ={ s=u*t }
```

Bi kasu bereizi behar ditugu: for exekutatzen denekoa eta exekutatzen ez denekoa.

Batean nahiz bestean, bukaeran ψ ondoko baldintza beteko da. Beraz:

$\{ t \geq 1 \wedge s=0 \}$ for i:=1 to t do s:=s+u $\{ s=u*t \}$

eta

$\{ t=0 \wedge s=0 \}$ for i:=1 to t do s:=s+u $\{ s=u*t \}$

For-aren propietatea:

$P(i) = \{s=u*i\}$

Eta (aurre(i))-ri egokitua: $P(aurre(i)) = \{s=u*(i-1)\}$

Frogapen formala:

1.– $\{t \geq 0\}\ s:=0\ \{t \geq 0 \wedge s=0\}$ (A.A.)

2.– $(t \geq 0 \wedge s=0) \rightarrow ((t=0 \wedge s=0) \vee (t \geq 1 \wedge s=0))$

3.– $(t=0 \wedge s=0) \rightarrow (t<1 \wedge s=0 \wedge t=0)$

4.– $\{t<1 \wedge s=0 \wedge t=0\}$ for i:=1 to t do s:=s+u $\{s=0 \wedge t=0\}$ (FEEA)

5.– $(s=0 \wedge t=0) \rightarrow (s=u*t)$

6.– $\{t=0 \wedge s=0\}$ for i:=1 to t do s:=s+u $\{s=u*t\}$ (FEEA)

7.– $(1 \leq i \leq t \wedge s=u*(i-1)) \rightarrow (s+u=u*i)$

8.– $\{s+u=u*i\}\ s:=s+u\ \{s=u*i\}$ (A.A.)

9.– $\{1 \leq i \leq t \wedge s=u*(i-1)\}\ s:=s+u\ \{s=u*i\}$ 7,8, (ODE)

10.– $(t \geq 1 \wedge s=0) \rightarrow (t \geq 1 \wedge s=u*0)$

11.– $\{t \geq 1 \wedge s=u*0\}$ for i:=1 to t do s:=s+u $\{s=u*t\}$ 9, (FORE)

12.– $\{t \geq 1 \wedge s=0\}$ for i:=1 to t do s:=s+u $\{s=u*t\}$ 10,11, (ODE)

13.– $\{(t=0 \wedge s=0) \vee (t \geq 1 \wedge s=0)\}$ for i:=1 to t do s:=s+u $\{s=u*t\}$ 6,12, (DJE)

14.– $\{t \geq 0 \wedge s=0\}$ for i:=1 to t do s:=s+u $\{s=u*t\}$ 2,13, (ODE)

15.– $\{t \geq 0\}$ begin s:=0; for i:=1 to t do s:=s+u $\{s=u*t\}$ 1,14, (KPE)

4.20. *adibidea*: *Frogatu programa honek A[1..n] osozko array-aren elementurik txikiena kalkulatzen duela*:

begin $\qquad \phi = \{n \geq 1\}$

```
  m:= A [1]
        {m= A [1]}
  for i:= 2 to n  do
        if  A[i]<m then  m:= A[i]
```

end $\{\exists j(1 \leq j \leq n \ \wedge \ m = A[j]) \ \wedge \ \forall j(1 \leq j \leq n \rightarrow m \leq A[j])\} = \psi$

Lehenengo asignazioa exekutatu ondoren bi aukera dauzkagu:

$(m = A[1] \ \wedge \ n = 1) \ \vee \ (m = A[1] \ \wedge \ n \geq 2)$

Aurreneko aukeran for-a ez da exekutatzen,

$(m = A[1] \ \wedge \ n = 1) \rightarrow n \geq 2$

eta

```
  { n≥2 ∧ m=A[1] ∧ n=1 }
        for i:=2 to n do if A[i]<m then m:=A[i]
  { m=A[1] ∧ n=1},
```

gainera:

$(m=A[1] \ \wedge \ n=1) \rightarrow \exists j(1 \leq j \leq n \ \wedge \ m=A[j]) \ \wedge \ \forall j(1 \leq j \leq n \rightarrow m \leq A[j]) = \psi$

eta, ondorioaren erregela erabiliz:

```
  { m=A[1] ∧ n=1 }
        for i:=2 to n do if A[i]<m then m:=A[i]
  {ψ}
```

Bestalde, for-a exekutatzen denean, hauxe da frogatu beharrekoa:

$\{n \geq 2 \ \wedge \ m = A[1]\}$ for $i := 2$ to n do if $A[i] < m$ then $m := A[i]$ $\{y\}$

baina horretarako nahikoa da

$P(i) = \exists j(1 \leq j \leq i \ \wedge \ m = A[j]) \ \wedge \ \forall j(1 \leq j \leq i \rightarrow m \leq A[j])$ propietatea

erabiliz $\{2 \leq i \leq n \wedge P(\text{aurre}(i))\}$ if $A[i] < m$ then $m := A[i]$ $\{P(i)\}$ frogatzea.

For-downto agindua (FORE)

For-to aginduarentzat ikusi dugun ideiaren haritik, for-downto aginduaren kasuan ere:

{true} for i:=10 downto 1 do A[i]:=0 $\left\{\forall j\left(1 \le j \le 10 \rightarrow A[j]=0\right)\right\}$ egiaztatzeko, aski da:

$\left\{1 \le i \le 10 \ \wedge \ \forall j\left(i < j \le 10 \rightarrow A[j]=0\right)\right\}$ $A[i]:=0$ $\left\{\forall j\left(i \le j \le 10 \rightarrow A[j]=0\right)\right\}$

frogatzea.

Honela bada, inferentzi erregelak ez du aparteko aldaketarik for-to aginduarekiko, i-ra iristeko zentzua izan ezik: for-to aginduan i-ren aurreko balioarentzat P propietatea bete eta ondoren i-rentzat betetzen zen bezalaxe, for-downto aginduan i-ren ondorengoarentzat betetzen da P lehenbizi eta gorputza exekutatu ostean i-rentzat.

$$\frac{\{n_1 \le i \le n_2 \ \wedge \ P(\text{ondoko}(i))\} \ \ I \ \{P(i)\}}{\{n_1 \le n_2 \ \wedge \ P(\text{ondoko}(n_2))\} \ \underline{\text{for}}\ i:=n_2 \ \underline{\text{downto}}\ n_1 \ \underline{\text{do}} \ \ I \ \{P(n_1)\}}$$

Gogoan izan behar da:

1.- $n_1 > n_2$ denean for-a ez dela exekutatzen eta aurreko baldintza kontserbatu egiten dela:

$\left\{n_1 > n_2 \ \wedge \ \phi\right\}$ for i:=n_2 downto n_1 do I $\left\{\phi\right\}$ (FEEA)

2.- Ik ez duela i aldagaia aldatzen, ezta n_1 eta n_2 adierazpenak ere.

3.- n_2-ren ondokoa definitu gabe egon daitekeela.

Erregistroen eremuei egindako asignazioa eta WITH sententzia (WTE)

Ikusi dugu nola array-en osagaiak ezin daitezkeen aldagai sinpletzat hartu.

Orain, erregistroen kasuan ez da gauza bera gertatzen, eremuak identifikadore batez unibokoki erreferentziatzen bait dira.

```
var x:  record
          a: integer,
          b: real,
          c: boolean
        end;
```

emanda, modu bakarra dago "x.a", "x.b" eta "x.c" eremuak aipatzeko eta guztiz esplizitua da.

Izen errepikatuzko kabiaketak daudenean ere:

```
var x:  record
            a: integer,
            b:record
                  a: integer;
                  b: boolean
              end;
            c: booloean;
        end;
```

erraz bereiz daitezke "x.b.a" eta "x.a", edo "x.b.b" eta "x.b".

With sententzia erabiliz gero eremu bat modu desberdinez erreferentziatu daiteke, eta honelako erabilerak arazo semantikoak sor ditzake.

Azken batean, with sententziak ordezkaketa sintaktikoa besterik ez du eragiten.

Erazagupen hau emanda:

```
var x:  record
            s1: T1;
            …
            sn: Tn
        end;
```

eta *with r do I* agindua, Iko edozein "r.s_i" osagai "s_i" erabiliz erreferentziatu daiteke besterik gabe. *with r do I* aginduaren semantika hauxe litzateke: "*with r do I* aginduak $r.s_1$, $r.s_2$,... $r.s_n$ eremuen gainean duen eragina Ik s_1, s_2,... ,s_n eremuen gainean, hurrenez hurren, duenaren parekoa da". Beraz, ondoko inferentzi erregela ondoriozta daiteke:

$$\frac{\{\phi\}\ I\ \{\psi\}}{\{\phi^{\,r.s_1,r.s_2,\ldots,r.s_n}_{s_1,s_2,\ldots,s_n}\}\ \underline{\text{with}}\ r\ \underline{\text{do}}\ I\ \{\psi^{\,r.s_1,r.s_2,\ldots,r.s_n}_{s_1,s_2,\ldots,s_n}\}}$$

Baina, lehen ikusi dugunez, with aginduak arazo semantikoak sor ditzake eta gerta liteke aurreko inferentzi erregela zuzena ez izatea. Arazo hauek erregistroak eta dagozkien with sententziak kabiatzen direnean sortzen dira, izen berdineko sekzio kabiatuak daudenean, hain zuzen ere. Ikus dezagun, adibide baten laguntzaz, zein den arazoa:

4.21. adibidea: *Izan bedi ondorengo erazagupena:*

```
var platera: record
        izena: packed array [1..n] of char;
        kaloriak: integer;
        osa1: record
                izena: packed array [1..n] of char;
                portzentaia, kaloriak: integer
        end;
 end;
```

Jo dezagun, orain, platera hasieratzeko agindu sinple hau daukagula:

```
with platera do
 begin
        izena:= "MARMITAKOA"; kaloriak:= 6;
        with osa1 do
                begin
                  izena:= "HEGALUZEA"; portzentaia:= 40; kaloriak:= 4
                end
 end;
```

garbi ikus daiteke ezin dela frogatu formalki edozein hasierako egoeratatik bukaerako egoera honako hau dela:

platera= ("MARMITAKOA",6, ("HEGALUZEA", 40,4)).

Honen zioa nabarmena da:

"platera.izena" eta "platera.osa1.izena" identifikadoreen artean talka gertatzen da eta berdin-berdin "platera.kaloriak" eta "platera.osa1.kaloriak"-en artean ere. Hortaz, bigarrengo with-aren gorputzean izen berdina daukate "izena" eta "kaloriak" eremuek.

Erregistroen eremuei egindako asignazioak, erregistro kabiatu, sekzio-izen errepikatu eta with aginduak konbinatzen direnean, arazo semantikoak sortzen ditu, nahiz eta with sententziaren semantika ordezkaketa sintaktikoa besterik ez den. Ez dago esan beharrik horrelakoak programaketan erabiltzea ez dela batere komenigarria, programen iluntasuna bait dakar.

Hala ere, inferentzi erregela mantendu egingo dugu, hori bai, erregistroen kabiaketa eta with sententziaren erabilera okerrik egiten ez den kasuetara murriztuz.

Fitxategiak eta sarrera/irteerako aginduak

"var f: file of T" erazagupenak $f \in h(T)*$ dela adierazten du, eta f maneiatzeko ondoko notazioa dugu hautatua:

$$f = \vec{f} \bullet \vec{f}, \text{ non } f\uparrow = \text{lehena}\left(\vec{f}\right)$$

Notazio hau erabiliz, hain zuzen, fitxategien gaineko oinarrizko aginduak hiru aldagai hoiei eginiko aldibereko asignazioen baliokideak gertatzen dira (2.4. atalean zabal jorratzen da ikusmolde hau). Ondoren aipatuko ditugun fitxategien gaineko oinarrizko aginduen axiomak baliokidetza horietan oinarritzen dira.

GET-aren axioma (GETA)

"get (f)" ezin daiteke exekutatu fitxategiaren azterketa bukatu denean:

$$\left\{ \vec{f} \neq <> \ \wedge \ \phi \begin{matrix} \vec{f} \bullet <f\uparrow>, \text{ hondar}\left(\vec{f}\right), \text{ lehen } \left(\text{hondar}\left(\vec{f}\right)\right) \\ \vec{f}, \vec{f}, f\uparrow \end{matrix} \right\} \ \text{get}(f) \ \{\phi\}$$

PUT-aren axioma (PUTA)

"put (f)" fitxategiaren bukaeran baizik ezin daiteke exekutatu:

$$\left\{ \vec{f} = <> \ \wedge \ \phi \begin{matrix} \vec{f} \bullet <f\uparrow>, \text{ indef} \\ \vec{f}, f\uparrow \end{matrix} \right\} \ \text{put}(f) \ \left\{\phi \wedge \ \vec{f} = <>\right\}$$

RESET-aren axioma (RSTA)

"reset (f)"-k ez du murriztapenik:

$$\left\{ \phi \begin{matrix} <>, \vec{f} \bullet \vec{f}, \text{lehen } \left(\vec{f} \bullet \vec{f}\right) \\ \vec{f}, \vec{f}, f\uparrow \end{matrix} \right\} \ \text{reset}(f) \ \{\phi\}$$

REWRITE-aren axioma (RWRA)

"rewrite (f)" aginduak f fitxategia hutsik uzten du, baina ez du buffer aldagaia aldatzen:

$$\left\{ \phi \begin{matrix} <>, <> \\ \vec{f}, \vec{f} \end{matrix} \right\} \ \text{rewrite}(f) \ \{\phi\}$$

EOF baldintza

"eof (f)", azken finean, $\vec{f} = <>$ baldintza boolearraren baliokidea da.

Sarrera/irteerako agindu nagusiak fitxategien gaineko irakurketa eta idazketa dira, ondoko baliokidetzak betetzen dituztela:

read(f,x) = begin x:= $f\uparrow$; get(f) end

write (f,x)= begin $f\uparrow$:= x; put (f) end,

beraz, (GETA) eta (PUTA), hurrenez hurren, eta konposaketaren erregela erabiliz egiaztatuko ditugu.

Orain arte esandako guztia input eta output fitxategientzat ere baliagarria da, ondoko bi berezitasun hauek kontuan hartuta:

- Input-aren gainean *reset, get, eof eta read* aginduak egikari daitezke.
- Output-aren gainean *put, write edo rewrite* soilik.
- Fitxategiaren izena jarri gabe uzten den aginduetan, input dela suposatzen da zenbait kasutan:

  reset = reset (input)
  eof = eof (input)
  read (x) = read (input, x)

  eta output dela beste kasu batzuetan:

  write (x) = write (output,x)
  rewrite = rewrite (output).
- Input-aren gaineko reset eragiketa inplizitua da (beste edozein baino lehenago egikaritzen da automatikoki), eta gauza bera gertatzen da rewrite eragiketarekin output-aren gainean.

Ikus ditzagun fitxategiak darabiltzaten programen zuzentasun partzialeko frogapenak.

4.22. adibidea: *Frogatu ondokoa:*

begin $\left\{\overleftarrow{f}=<1,3,5,\ldots,2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$

n:= n+1;

write (f, 2∗n+1)

end $\left\{\overleftarrow{f}=<1,3,5,\ldots,2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$

1.– $\left(\overleftarrow{f}=<1,3,5,\ldots,2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right)\rightarrow$

$\left(\overleftarrow{f}=<1,3,5,\ldots,2(n+1)-1> \wedge \vec{f}=<> \wedge n+1\geq 0\right)$

2.– $\left\{\overleftarrow{f}=<1,3,5,\ldots,2(n+1)-1> \wedge \vec{f}=<> \wedge n+1\geq 0\right\}$

$n:=n+1$

$\left\{\overleftarrow{f}=<1,3,5,\ldots,2n-1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$ (A.A.)

3.– $\left\{\overleftarrow{f}=<1,3,5,\ldots,2n-1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$

$f\uparrow:=2*n+1$

$\left\{\overleftarrow{f}=<1,3,5,\ldots,2n-1> \wedge \vec{f}=<> \wedge n\geq 0 \wedge f\uparrow:=2*n+1\right\}$ (A.A.)

$\left\{\overleftarrow{f}=<1,3,5,\ldots,2n-1,\ 2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$ (PUTA)

4.– $\left\{\overleftarrow{f}=<1,3,5,\ldots,2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$

begin $n:=n+1$; write $(f,\ 2*n+1)$ end

$\left\{\overleftarrow{f}=<1,3,5,\ldots,2n+1> \wedge \vec{f}=<> \wedge n\geq 0\right\}$ 1,2,3, (PUTA), (ODE) eta (KPE)

***4.23. adibidea**: Frogatu ondoko programak f fitxategia irakurtzen duela txuriguneren bat aurkitu edo, txurigunerik ez badago, bukaerara iritsi arte:*

```
begin
        reset (f); read (f,x);
        while (not eof (f)) and (x< > ' ') do read (f,x)
end;
```

Frogapenaren eskema honako hau da:

begin $\left\{f=<k_1,k_2,\ldots k_n> \wedge n\geq 1\right\}$

reset (f);

$\left\{\overleftarrow{f}<> \wedge \vec{f}=<k_1,k_2,\ldots k_n> \wedge f\uparrow = k_1\right\}$

read (f,x);

$\left\{\overleftarrow{f}=<k_1> \wedge \vec{f}=<k_2,\ldots k_n> \wedge\ f\uparrow = k_2 \wedge x=k_1\right\}\rightarrow$

$INB=\left\{\exists i\left(1\leq i\leq n \wedge x=k_i \wedge \overleftarrow{f}=<k_1,k_2,\ldots,k_i> \wedge \vec{f}=<k_{i+1},\ldots k_n> \wedge\right.\right.$

$\left.\left. f\uparrow = k_{i+1} \wedge \forall j\left(1\leq j<i\rightarrow k_j\neq' '\right)\right)\right\}$

while (not eof (f)) and (x< >' ') do

read (f,x)

end;

$\left\{\exists i\left(1\leq i\leq n \wedge x=k_i \wedge \overleftarrow{f}=<k_1,k_2,\ldots,k_i> \wedge \vec{f}=<k_{i+1},\ldots k_n> \wedge\right.\right.$

$\left.\left.\forall j\left(1\leq j<i\rightarrow k_j\neq' '\right) \wedge \left(k_i=' ' \vee\ i=n\right)\right)\right\}$

Frogapen formalak badu azpimarragarria den aspektu tekniko bat. Inbariantea kontserbatzen dela frogatzean lortzen den ondoko baldintza ez da inbariantea bera, baizik eta formula baliokide bat:

$$\left(\text{INB} \wedge \neg\, \text{eof}(f) \wedge x \neq ' \,'\right) \rightarrow$$
$$\left(\exists i \left(1 \le i < n \wedge x = k_i \wedge \overleftarrow{f} = < k_1, k_2, \ldots, k_i > \right.\right.$$
$$\left.\left. \wedge\, \vec{f} = < k_{i+1}, k_{i+2}, \ldots, k_n > \wedge\ f\uparrow = k_{i+1} \wedge \forall j \left(1 \le j \le i \rightarrow k_j \neq ' \,'\right)\right)\right)$$

Ondorioz:

$$\left\{\exists i \left(1 \le i < n \wedge x = k_i \wedge \overleftarrow{f} = < k_1, k_2, \ldots, k_i > \right.\right.$$
$$\left.\left. \wedge\, \vec{f} = < k_{i+1}, k_{i+2}, \ldots, k_n > \wedge\ f\uparrow = k_{i+1} \wedge \forall j \left(1 \le j \le i \rightarrow k_j \neq ' \,'\right)\right)\right\}$$

read (f,x)

$$\left\{\exists i \left(1 \le i+1 \le n \wedge x = k_{i+1} \wedge \overleftarrow{f} = < k_1, k_2, \ldots, k_{i+1} > \right.\right.$$
$$\left.\left. \wedge\, \vec{f} = < k_{i+2}, k_{i+3}, \ldots, k_n > \wedge\ f\uparrow = k_{i+2} \wedge \forall j \left(1 \le j < i+1 \rightarrow k_j \neq ' \,'\right)\right)\right\}$$

Bistan denez, lortutako ondoko baldintza inbariantearen baliokidea da: inbariantea $\exists i(\phi)$ erakoa da eta ondoko baldintza, berriz, $\exists i\ (\phi)_i^{i+1}$ itxurakoa. Bi formula hauek, aldibereko ordezkaketaren semantika gogoan hartuta, baliokideak dira.

Programa honetan, fitxategiak tratatzen dituzten beste hainbatetan bezalaxe, borne-adierazpena tratatzeke dagoen fitxategiaren luzeraren arabera asma daiteke. Pauso bakoitzean gutxienez elementu bat tratatzen bada, adierazpen hori beheratu egiten da $(\mathbb{N}, \le)$ ondo oinarritutako ordenan.

Fitxategiaren luzera aztertzeko *luz* funtzioa definituko dugu:

$$\text{luz: } T^* \rightarrow T, \text{ non luz } (<>)=0 \text{ eta luz } \left(< a_1, a_2, \ldots, a_n >\right) = n$$

Gure adibiderako har dezagun: $e = \text{luz}\left(\vec{f}\right)$

a) $\left(\text{INB} \wedge \neg\, \text{eof}(f) \wedge x \neq ' \,' \rightarrow \vec{f} \neq <>\right) \rightarrow \left(\text{luz}\left(\vec{f}\right) > 0\right) \rightarrow \left(\text{luz}\left(\vec{f}\right) \in \mathbb{N}\right)$

b) $\left\{0 < \text{luz}\left(\vec{f}\right) = b\right\}\ \text{read}(f, x)\ \left\{0 \le \text{luz}\left(\vec{f}\right) < b\right\}$

Pauso-kopurua dela eta, adierazpen honek kasurik txarrena bornatzen du. Zero balioa hartuko badu txurigunerik aurkitu gabe edo bestela, txurigunerik aurkitzekotan, fitxategiaren azken posizioan aurkituz amaitu behar du iterazioaren exekuzioak.

Ariketak

4.16. Froga ezazu ondoko programak A[1..n] eta B[1..n] osozko array-en bidez errepresentatutako bektoreen biderkadura eskalarra kalkulatzen duela:

```
begin
    p:=0;
    for k:=1 to n do
        p:=p+ A[k] * B[k]
end
```

4.17. Frogatu ondoko programak A[1..n] eta B[1..n] osozko array-etan errepresentatutako bektoreen biderkadura eskalarra kalkulatzen duela:

```
begin
    p:=0;
    for k:=1 to n do p:=p+A[k]*B[k]
end
```

4.18. Ondoko programa espezifikatua emanda, frogatu zuzena dela:

```
cons    n = ?;   { n ≥ 0 }
var   x,z,i: integer;
          A: array [0..n] of integer;
begin
    z := A[n];
    for i:=1 to n do z:= A[n-i]+z*x;
end   { z = A[0] + A[1]*x + A[2]*x^2 + ..... + A[n]*x^n }
```

4.19. Ondorengo programak A[1..n] array-a alderanztu egiten du. Dokumenta ezazu:

```
begin     ϕ={
    for i:=1 to n div 2 do   P(i)= {
        begin
            z:=A[i];
            A[i]:=A[n-i+1];
            A[n-i+1]:=z
        end
end   ψ={
```

4.20. Input-ean zenbaki negatiborik bada, azken zenbaki negatiboa idazten duen programa honen zuzentasun osoa froga ezazu:

```
var A: array[1..n] of integer;
    begin
        for i:=1 to n do read (A[i]);
        j:=n;
        while (j>1 and A[j] >=0) do j:=j-1;
        if A[j] < 0 then write (A[j])
            else write ('ez dago negatiborik')
end
```

4.21. Ondoko programak input-eko lehen elementua bigarrenaren berredura den ala ez idazten du biak zenbaki arrunt positibo direnean, eta baiezkoan (lehena bigarrenaren berredura denean) berretzailea ere idazten du. Dokumenta ezazu.

```
var x, y, z : integer;
    begin
        read (x); write (x); read (z);
        if (x<=0 or z<=0) then write (z)
        else
            begin
                y:=0;
                while (x>1 and x mod z = 0) do
                    begin x := x div z; y := y+1 end;
                if x=1 then write (z, 'ber', y, 'da')
                    else write ('ez da', z, 'ren berredura')
            end
    end.
```

4.22. Ondorengo programak **f** eta **g** fitxategi ordenatuetako edukia **h** fitxategian idazten du ordenari eutsiz. Dokumentatu eta zuzentasuna egiaztatu.

```
begin       {     }
        reset (f); reset (g); rewrite (h);
        while not (eof(f)) and not (eof(g)) do    INB = {     }
            begin
                if f↑<g↑ then
                    begin h↑:=f↑; get (f) end
                 else
                    begin h↑:=g↑; get (g)
                    end;
                put (h)
            end;
          {       }
        while not (eof (f)) do
            begin
                h↑:=f↑; put (h); get (f)
            end;
          {       }
        while not (eof (g)) do
            begin
                h↑:=g↑; put (h); get (g)
            end;
end     {         }
```

4.23. Programa honek A[1..n, 1..m] array-a txikienetik handienera ordenatuta dagoen ala ez idazten du, ordena hori zutabeka eta zutabe bakoitzean goitik behera hartuz. Dokumenta ezazu.

```
var i, j: integer; b: boolean; A: array[1..n,1..m] of integer;
    begin
        i:=1; j:=1; b:=true;
        while (i<n or j<m) and b do
        begin
            if i=n then begin  b:=(A[i,j] <= A[1,j+1] ); i:=1; j:=j+1 end
            else begin b:=(A[i,j] <= A[i+1,j] ); i:=i+1 end
        end;
        if b then write ('zutabeka gorakorra da')
        else write (A[i,j] , 'goraka ez dagoen lehen elementua da')
    end.
```

Parametrorik gabeko prozedurak (PGPE)

Aldagai orokorretan ez bestetan eragiten duen parametrorik gabeko prozedura bati dei egitea, prozedura horren gorputza exekutatzearen parekoa da. Aldiz, prozeduraren gorputzak aldagai lokalen batean eraginik badu, deiaren kontrola galtzen denean, aldagai horrek ez du inolako zentzurik, ezta balio definiturik ere. Ondorioz, aldagai lokalekiko dagoen aldea dela eta, ezin daiteke esan deiaren eragina eta gorputzaren exekuzioaren eragina guztiz baliokideak direnik.

Izan bedi ondoko prozedura:

```
var x,y, lagun: integer;
procedure TRUKATZE;
begin
    lagun:=x;
    x:=y;              (* agertzen diren aldagai guztiak orokorrak dira *)
    y:=lagun
end;
```

zeinek espezifikazio hau betetzen duen:

$\{x = a \wedge y = b\}$ begin lagun:=x; x:=y; y:=lagun end; $\{x = b \wedge y = a \wedge lagun = a\}$

Honako hau ondoriozta dezakegu, beraz:

$\{x = a \wedge y = b\}$ TRUKATZE $\{x = b \wedge y = a \wedge lagun = a\}$

Har dezagun beste programa hau, ordea:

```
var x,y: integer;
procedure TRUKATZE;
var lagun: integer;
begin
    lagun:=x;
    x:=y;
    y:=lagun
end;
```

eta ondoko zuzentasun partzialari buruzko baieztapena:

$\{x = a \wedge y = b\}$ begin; $lagun = x;\ x := y;\ y := aux$ end $\{x = b \wedge y = a \wedge lagun = a\}$

Ezin dezakegu $\{x = a \wedge y = b\}$ TRUKATZE $\{x = b \wedge y = a \wedge lagun = a\}$ inferitu, *lagun* aldagaia TRUKATZE prozedurari eginiko deia bukatzen denean indefinituta geratzen bait da.

Beraz, honelako parametrorik gabeko prozedura bat edukiz gero:

procedure R;

S ;

Sren zuzentasun partzialari buruzko baieztapenetan aldagai orokorrak baizik agertzen ez badira, baieztapen hauek betetzen dira Rri egindako deientzat ere. Baina baieztapenetan aldagai lokalak ere ageri badira, deiarentzat eginiko inferentzia ez da zuzena izango eskuharki.

Hona bada, inferentzi erregela:

$$\frac{\{\phi\}\ S\ \{\psi\}}{\{\phi\}\ R\ \{\psi\}}\ \textbf{(*1)}$$

(*1): baldin ϕ eta ψ -n libre ageri diren aldagai guztiak orokorrak badira.

Normalean prozedura bati deitzen zaionean, prozedurak erabili edo aldatuko dituen objektuez gain beste batzuek ere definitzen dute konputazio-egoera. Beste objektu hauek programa printzipalarentzat garrantzitsuak diren propietateak betetzen dituzte, ez dira hala ordea deiarentzat. Prozedurak aldatzen ez duen guztia kontserbatu egiten delako ideia formalizatzeko, axioma bat eta erregela bat definituko ditugu.

Kontserbazioaren axioma (K.A.)

Programa batek (I) ez badu eraginik aurreko baldintzako aldagai libreetan, hots, ez baditu aldatzen, orduan aurreko baldintza kontserbatu egiten da:

$$\{\phi\}\ I\ \{\phi\}$$

baldin I-k ez badu eraginik ϕ-ko ezein aldagai libretan.

Axioma hau guztiz orokorra da, baina gurera etorriz, axiomako I identifikadorea egiaz prozedura-deia denean, eta dei horrek ez baditu programako aldagai guztiak aldatzen, bere horretan irauten duten aldagaien propietateak mantendu egiten dira deiaren ondoren.

Konjuntzioaren erregela (K.J.E.)

Aurreko axioma erabiltzeko aukera ematen du erregela honek, deiak kontserbatu eta kontserbatzen ez dituen zatiak bereiziz aurreko baldintzan:

$$\frac{\{\phi_1\}\ I\ \{\psi_1\},\ \{\phi_2\}\ I\ \{\psi_2\}}{\{\phi_1\}\ \wedge\ \{\phi_2\}\ I\ \{\psi_1 \wedge \psi_2\}}$$

4.24. adibidea: *Frogatu ondoko programak B[1..m] array-an uzten duela A[1..n,1..m] array-eko zutabe bakoitzaren baliorik handiena.*

```
cons      n= ; m= ;
var       A: array [1..n, 1..m] of integer;
          B: array [1..m] of integer;
          j, hand:  integer;
procedure          ZUTABE_HANDI_J;
var       i: integer;
begin               {1≤j≤m}
          i:=1;
          hand:=A[i,j];
                    P(i) = { hand= max (A[1..i, j] }
          for i:=2  to n do
                    if  A[i,j] > hand then hand:=A [i,j]
          end       {hand= max (A [1..n,j])}
begin               {true}
 j:=1;
          INB = {1≤j≤m+1  ∧  ∀k(1≤k<j→B[k]=max (A[1..n,k]))}
 while j ≤ m do
 begin
          ZUTABE_HANDI_J;
          B[j]:= hand;
          j:=j+1
end
end                 {∀k(1≤k≤m→B[k]=max (A[1..n,k]))}
```

non x=max (D[1..a,b]) laburdura erabili bait da D array-an b zutabeko 1etik a errenkadarainoko elementuen maximoa x dela adierazteko, hau da, formula honen ordez:

$$\{\forall i(1 \le i \le a \rightarrow x \ge D[i,b]) \wedge \exists i(1 \le i \le a \wedge x = D[i,b])\}$$

(KA) eta (KJE)-ekin batera (PGPE) erregelaren erabilera ilustratzeko, programa nagusiaren inbariantea kontserbatu egiten dela frogatuko dugu formalki:

1.– $\left\{\forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right\}$

ZUTABE_HANDI_J (* B eta J ez dira aldatzen prozeduran *)

$\left\{\forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right\}$ (KA)

2.– $\left\{1 \le j \le m\right\}$ ZUTABE_ HANDI_ J $\left\{hand = \max\left(A[1..n,j]\right)\right\}$ (PGPE)

3.– $\left\{1 \le j \le m \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right\}$

ZUTABE_HANDI_J

$\left\{1 \le j \le m \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right) \wedge hand = \max\left(A[1..n,j]\right)\right\} 1,2,$ (KJE)

4.– $\left\{1 \le j \le m \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right) \wedge hand = \max\left(A[1..n,j]\right)\right\}$

B[j]:=hand

$\left\{1 \le j \le m \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right) \wedge B[j] = \max\left(A[1..n,j]\right)\right\}$ (A. A.)

5.– $\left(1 \le j \le m \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right) \wedge B[j] = \max\left(A[1..n,j]\right)\right) \rightarrow$

$\left(1 \le j \le m \wedge \forall k\left(1 \le k \le j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right) \rightarrow$

$\left(1 \le j+1 \le m+1 \wedge \forall k\left(1 \le k < j+1 \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right)$

6.– $\left\{1 \le j+1 \le m+1 \wedge \forall k\left(1 \le k < j+1 \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right\}$

j:=j+1

$\left\{1 \le j \le m+1 \wedge \forall k\left(1 \le k < j \rightarrow B[k] = \max\left(A[1..n,k]\right)\right)\right\}$ (A. A.)

7.– $\left\{INB \wedge j \le m\right\}$ I $\left\{INB\right\}$ 3, 4, 5, 6, eta (KPE)

Prozedura parametrodunak (PPE)

Prozedura parametroduna izatez ekintza bat da, datu gisa aldagai generiko batzuk hartuz (baliozko parametro formalak), emaitza gisa beste aldagai generiko batzuk lortzen dituena (aldagaizko parametro formalak). Gainera, programaren egituraketa eta argitasunaren mesedetan, komeni da ekintza horrek berez zentzua edukitzea, hau da, prozeduraren eragina ahalik eta isolatuena egotea.

Prozedura parametrodun batek aldagai orokorren bat erabili eta ez aldatzeak, datu gisa erabiltzen duela adierazten du, hots, baliozko parametro gisa, eta honelako erabilerak esplizituki ez adierazteak deien semantiken nahastea eta programaren ulertezintasuna ekar lezake. Are kaltegarriagoa da prozedura parametrodunak albo-ondorioak edukitzea, hots, aldagai orokorren bat aldatzea; izan ere honelakoetan erreferentziazko parametroetan ezezik, inplizituki, beste emaitzak ere lortzen bait dira.

Programazio-metodologiari dagokionez orain arte aipatutakoak ohitura txarrak baizik ez dira, diseinu argiko eta ondo egituratutako programa zuzenen optimizaziorako ez bada, justifika ezin daitezkeenak. Nolanahi ere, programatzaileak beti izan behar du gogoan honelako zailtasun semantikoek behar bezalako dokumentazioa eskatzen dutela.

Prozedura parametrodunek aldagai lokalak ere eduki ditzakete, egin beharrekoa burutzeko aldagai laguntzaile gisa erabiliko direnak. Are, prozedura parametrodun batek aldagai laguntzaileak behar baditu, hauek lokalak izan behar dute.

4.25. adibidea: *Ondoko programak A[1..n] array-a (aldagai orokorra) ordenatu egiten du hautaketa bidez. Metodo honetan A[1..n]-ko elementurik txikiena aurkitu eta lehenengoarekin trukatzen da, ondoren A[2..n]-ko txikiena bigarrengoarekin, eta gisa honetako trukaketak array-a erabat ordenatu arte burutzen dira.*

```
var       A: array [1..n] of integer;
          j,k: 1..n;
procedure          TRUKATZE (var x,y: integer);
var       lagun: integer;
begin              {x=a ∧ y=b}
          lagun:=x; x:=y; y:=lagun
end                {x=b ∧ y=a}
procedure TXIKIENA (B: array [1..n] of integer; p:integer; var i:1..n);
var       (* i aurkitzeko aldagai lagungarriak *)
begin     {1≤p≤n}
  ...
end       {∀r(p≤r≤n → B[r]≥B[i])}
begin              {A=(a1,...,an)}                (* programa nagusia *)
 j:=1;
          INB={A=(b1,b2,...,bn) ∧ A=perm(a1,a2,...,an) ∧
               1≤j≤n ∧ gorakor(A[1..j-1]) ∧ txikiagoak(A,j-1,n)}
 while j<n do
 begin
          TXIKIENA (A,j,k);
          γ1={A=(b1,b2,...,bn) ∧ A=perm(a1,a2,...,an) ∧
              1≤j<n ∧ gorakor(A[1..j-1]) ∧ txikiagoak(A,j-1,n) ∧
              j≤k≤n ∧ ∀r(j≤r≤n → A[r]≥A[k])}
          TRUKATZE (A [j], A[k]);
          γ2={A=perm(a1,a2,...,an) ∧ gorakor(A[1..j]) ∧
              txikiagoak(A,j,n) ∧ 1≤j<n}
          j:= j+1
 end
end                {A=perm(a1,a2,...,an) ∧ gorakor(A[1..n])}
```

non, argitasuna dela zio, asertzioak laburtzapenak erabiliz idatzi bait dira:

$$\texttt{gorakor}(\texttt{A}[\texttt{b..g}]) \leftrightarrow \forall \texttt{i}(\texttt{b} \le \texttt{i} < \texttt{g} \rightarrow \texttt{A}[\texttt{i}] \le \texttt{A}[\texttt{i}+1])$$
$$\texttt{txikiagoak}(\texttt{A,b,g}) \leftrightarrow \forall \texttt{i} \forall \texttt{h}(1 \le \texttt{i} \le \texttt{b} \ \wedge \ \texttt{b} < \texttt{h} \le \texttt{g} \rightarrow \texttt{A}[\texttt{i}] \le \texttt{A}[\texttt{h}])$$

Prozedura parametrodun bati eginiko deiaren semantika, parametro formalak parametro errealez ordezkatuta burututako prozedura horren exekuzioa da, betiere aldagai orokorretan eraginik ez duen kasuetan. Esan daiteke, hortaz, deiak bere baitan dauden parametro errealetan duen eragina eta prozeduraren gorputzak parametro formaletan duen eragina berdin-berdinak direla. Prozeduran aldagai orokorrak agertzen badira, ordea, parametro erreal gisa erabiltzen diren aldagai orokorren eta albotik aldatzen diren aldagai orokorren arteko erlazioak baldintzatzen du prozeduraren eragina eta, ondorioz, oso zaila gertatzen da deiaren semantika definitzea.

Ikus dezagun adibide sinple batez aipatutako arazo hau. Izan bedi ondoko prozedura, x aldagai orokorrean albo-ondorio bat eragiten duena:

```
var         x,y,z: integer;
procedure ERAGINA (datu: integer; var balabs: integer);
begin       {true}
            if datu <0 then
                        begin
                                    balabs:=–datu;
                                    x:= datu
                        end
            else
                        balabs:= datu
end         {balabs =|datu|  ∧  (datu<0 → x=datu)}
```

Zalantzarik gabe taxuzko espezifikazio bat eman dugu eta bertan aldagai orokorra ageri da, bestela ezinezkoa bait litzateke albo-ondorioa adieraztea. Espezifikazio honetan parametro formalak parametro errealez ordezkatuta deiaren eragina adierazten duen Hoare-ren hirukotea lortzen da, baina hirukote hau zuzena edo okerra izan daiteke deiaren parametroen eta x aldagaiaren arteko erlazioaren arabera.

Adibidez, ERAGINA (x,y) deiarentzat ordezkaketa eginez gero:

$$\{\text{true}\}\ \text{ERAGINA}\ (x,y)\ \{y = |x| \wedge (x < 0 \rightarrow x = x)\}$$

lortzen da, eta hirukote horrek azken batean baieztatzen duena

{true} ERAGINA (x,y) {y= |x|} da, baieztapen zuzena, hortaz.

ERAGINA (y,x) deiarentzat, berriz, baieztapen hau lortzen da:

$$\{\text{true}\}\ \text{ERAGINA}\ (y,x)\ \{x = |y| \ \wedge \ (y < 0 \rightarrow x = y)\}$$

ondoko baldintza hori beste era honetaz ere adieraz daiteke:

$$(y \geq 0 \rightarrow x = y) \wedge (y < 0 \rightarrow x = -y) \wedge (y < 0 \rightarrow x = y),$$

eta formula hori faltsua denez, hasierako baieztapena honela geratuko zaigu:

{true} ERAGINA (y,x) {false},

faltsua da, beraz, hasierako baieztapena.

Ondorioz, *procedure P (*$\overline{x}$*:< mota-eraz>); S;* prozeduraren egiaztapen-erregela (semantika formala) ezartzerakoan Sn aldagai orokorrik ez dela agertuko joko dugu.

Aldagai lokalei dagokienez, parametrorik gabeko prozedurentzat adierazitako ideiak balio du parametrodunentzat ere, hau da, ez dutela agertu behar ez aurreko baldintzan, ez ondoko baldintzan (prozedurak kontrola duen bitartean bait daude definituta eta, horretara, indefinituak dira deiaren aurretik eta ondotik).

Azkenik, erreferentziazko eta baliozko parametroekin gertatzen dena aztertzea ere interesgarria da. Dagoeneko badakigu prozedurari eginiko deiak erreferentziazko parametro errealak aldatzen dituela, prozeduraren gorputzak parametro formalak aldatzen dituen bezalaxe. Baina ez da horrela gertatzen baliozko parametroetan, aldatzen direnak kopiak bait dira, errealak ukigabe geratzen diren bitartean.

Demagun adibidez,

```
var u,k: integer;
procedure TRUKATZE (var x,y: integer);
var lagun: integer;
begin
        lagun:=x; x:=y; y:=lagun
end;
```

TRUKATZE (u,k) deiak u eta k aldagaietan duen eragina, prozeduraren gorputzak x eta y aldagaietan duenaren parekoa da, x eta y dauden memoriako posizioak aldatzen bait dira zuzen-zuzen.

Aitzitik, demagun:

```
var u,k: integer;
procedure TRUKATZE (x,y: integer);
var  lagun: integer;
begin
        lagun:=x;  x:=y; y:=lagun
end;
```

TRUKATZE (u,k) deia exekutatuz u eta k aldagaien kopiak (memoriako beste bi posiziotan egindakoak) aldatzen dira, deia bukatutakoan u eta k aldatu gabe, eta x eta y indefinituta geratzen direla. Hau da, x eta y erreferentziazko parametroak badira:

$$\{u = a \ \wedge \ k = b\} \text{ TRUKATZE } (u, k) \ \{u = b \ \wedge \ k = a\}$$

baieztapena zuzena da; baina x eta y baliozko parametroak badira baieztapen zuzena ondokoa da:

$$\{u = a \wedge k = b\}\ \text{TRUKATZE}\ (u,k)\ \{u = a \wedge k = b\}$$

Argi eta garbi ikusi denez, prozeduraren gorputzak baliozko parametro formaletan eragiten duen edozein aldaketa ezin daiteke deiarentzat eta bere parametro errealentzat inferitu.

Gorputzarentzat frogatutako baieztapenak deiarentzat ere baieztagarri izateko aukera emango digun inferentzi erregela bat planteatzean irtenbide bat hauxe liteke: ondoko baldintzan baliozko parametroak libre ager daitezen debekatzea. Kasu honetan, aurreko baldintzan hasierako balioak eman eta ondoko baldintzan hasierako balioak erabil ditzakegu, azken finean, semantikoki datuak baizik ez bait dira.

Baina programazio-metodologiari atxekituz aproposagoa da gorputzean baliozko parametroen aldaketak (baliozko parametroei eginiko asignazioak) gerta daitezen debekatzea, semantikoki ez bait dute inolako zentzurik.

Beraz, aztertutako arazoei emandako irtenbideak kontuan hartuta,

```
procedure R (x̄:<parametro-motak>);
S;
```

prozedurari eginiko $R(\bar{a})$ deiaren egiaztapen-erregela honelakoa izango da:

$$\frac{\{\phi\}\ S\ \{\psi\}}{\left\{\phi^{\bar{a}}_{\bar{x}}\right\}\ R(\bar{a})\ \left\{\psi^{\bar{a}}_{\bar{x}}\right\}}$$

ondoko murriztapenak bete behar direla:

-Sn ez dago aldagai orokorrik.

- ϕ eta ψ asertzioetan ez dago aldagai lokal librerik.

-Sn ez dago baliozko parametroren bati eginiko asignaziorik.

Baina aurrekoa ez da oraindik erregela zuzena. Bada erregela zertxobait aldaraziko duen beste arazo bat. Azter dezagun ondoko adibidea:

```
var h: integer;
procedure ADIBIDEA (var p: integer; i:integer);
        begin
                p:=i+1
        end;
```

Emandako erregela erabiliz, {true} begin p:=i+1 end {p=i+1} baieztapenetik {true} ADIBIDEA (h,h) {h=h+1} ondoriozta daiteke, eta baieztapen hau {true} ADIBIDEA (h,h) {false} hirukotearen baliokidea da. Beraz, deiari buruzko baieztapen faltsu bat inferitu dugu prozeduraren gorputzari buruzko egiazko baieztapen batetik. Aurreko erregela aldatu behar dugu nahitaez, ondoko definizioa kontuan hartuz:

Bereizte-baldintza: Izan bedi $\bar{y}$ Sn aldagarriak diren Rko parametro formalen zerrenda eta izan bedi $\bar{z}$ Sn aldatzen ez diren Rko parametro formalen zerrenda (hau da, $\bar{x} = \bar{y} \bullet \bar{z}$). Izan bitez $\bar{b}$ eta $\bar{d}$ dagozkien parametro errealen zerrendak ($\bar{a} = \bar{b} \bullet \bar{d}$). $\bar{a}$-k bereizte-baldintza betetzen duela esango dugu, $\underline{\text{bereiz}(\bar{a})}$, baldin eta soilik baldin $\bar{b}$-ko aldagai guztiak desberdinak badira eta $\bar{d}$-n ez bada $\bar{b}$-ko aldagairik ageri.

Honenbestez, hauxe da prozedura parametrodunen inferentzi erregela:

Lehenxeago ikusi dugun adibidean $\bar{y} = p$, $\bar{z} = i$, $\bar{b} = h$, $\bar{d} = h$, genuen eta bereiz (h,h) ez zen betetzen. Horrek esan nahi du PPE erregelaz ezin daitekeela ezer ondorioztatu ADIBIDEA (h,h) deiarentzat.

4.26. adibidea: *Input fitxategian errepresentaturik ditugun n tamainako bi bektoreen biderkadura eskalarra idazten duen ondoko programaren zuzentasuna azter ezazu, suposatuz adierazten diren prozeduren espezifikazioak betetzen direla:*

```
cons      n=?;     {n≥1}
var       A,B: array [1..n] of integer;
          balioa: integer;
procedure IRAKURRI (var D:array [1..n] of integer);
begin     {input⃗=<e1,e2,...em> ∧ m≥n}
  ...
end       {D=(e1,e2,...,en) ∧ input⃗=<en+1,...,em>}
procedure BIDER (P,K: array [1..n] of integer; var r:integer);
begin     {true}
  ...
end       {r=P*K}
begin     {input=<a1,a2,...,an,b1,b2,...,bn>}
  reset;  {input⃗=<a1,a2,...,an,b1,b2,...,bn>}
  IRAKURRI (A);          {A=(a1,a2,...,an) ∧ input⃗=<b1,b2,...,bn>}
  IRAKURRI (B);          {A=(a1,a2,...,an) ∧ B=(b1,b2,...,bn)}
  BIDER(A,B,balioa);{A=(a1,a2,...,an) ∧ B=(b1,b2,...,bn) ∧ balioa=A*B}
  write (balioa)
end;      {output=<(a1,a2,...,an)*(b1,b2,...,bn)>}
```

Frogapenaren eskema programarekin batera doa. Parametro errealen bereizketa nabaria da, eta deien arteko asertzioak prozeduren espezifikazioan parametro formalak errealez ordezkatuz eta kontserbazioaren axiomaz lortzen dira. Aspektu teknikorik nabarmentzekotan, komeni da hasierako balioak generikoak direla aipatzea eta gisa honetan hartu behar direla gogoan izatea frogapenean.

Funtzioak (FUNE)

Matematiketan, $f: A \rightarrow B$ funtzioa korrespondentzia bat da, non Ako elementu bakoitzari gehienez ere Bko elementu bakarra dagokion. Funtzio partziala dela esaten dugu Ako elementuren bati Bko elementurik ez dagokionean (bestela, totala dela esaten da).

Pascal-eko funtzioak erazagutzean izena, eremua eta heina adierazten dira:

function f $(\overline{x}: A)$: B;

eta edozein x-ri dagokion f(x) kalkulatzeko algoritmoaren definizioa, honela:

```
begin       {ϕ}     (* ϕ-n x̄-ren agerpena librea da *)
  ...
f:= ...
end;        {ψ}     (* ψ-n f libre ageri da, f(x̄) adieraziz *)
```

Aurreko baldintzak murriztapenak ezartzen ditu eta, funtzioa partziala bada (ez dago definitua A multzo osoarentzat), funtzioaren definizio-eremua finkatzen du. Ondoko baldintzak $\overline{x}$-ren araberako f aldagaiaren balioa definitzen du, non f aldagaiak, semantikoki, f($\overline{x}$) errepresentatzen duen.

Pascal-eko funtzio kontzeptua ez dator bat kontzeptu matematikoarekin, albo-ondorioak ere gerta litezkeenak bait dira. Honako hauek, esate baterako, maiztto ikusten dira:

- aldagai orokorren aldaketa.
- argumentuen aldaketa (erreferentziazko parametroak).

Ikus ditzagun Pascal-eko bi funtzio hauek:

```
var z:integer;
function f (x:integer): integer;
        begin
                z:=x+1;
                f:=z
        end;
function g (var x:integer): integer;
        begin
                x:=x+2;
                g:=z
        end;
```

bada, definizio hauen arabera, ezin daiteke baieztatu *f(z)+g(z)= g(z) + f(z)* gertatzen denik.

Agerikoa da albo-ondorioak baimenduz gero funtzio-deien semantika gehiegi aldrebesten dela. Programazio-metodologiari eta prozeduren albo-ondoioei buruz

esandakoak balio du funtzioentzat ere. Prozedurek bezala, funtzioek ere eduki ditzakete aldagai lokalak eta hauen semantika lehen bezala definitzen da, aldagai laguntzaileen betebeharra dutela.

Beraz, finkatuko dugun semantika formala (egiaztapen-formala) albo-ondoriorik gabeko funtzioetara zuzendurik dago.

Albo-ondoriorik gabeko ondoko funtzioa emanda,

function F $(\bar{x}$: <parametro – motak>): <emaitza – mota>;

S;

$F(\bar{a})$ deia ez da izango programa nagusiko agindua, emaitza-motako balioa duen adierazpena bait da. Eta, hain zuzen, adierazpen gisa agertuko da programa nagusian. Hara adibide batzuk:

$x := F(\bar{a})$;

if $F(\bar{a}) > 0$ then …;

$x := x + F(\bar{a})$;

etab.

Funtzioaren gorputzak $\{\phi\}$ S $\{\psi\}$ betetzen badu eta $\bar{x}$ parametroek ϕ aurreko baldintza egiazko egiten badute, irteeran F aldagaiak ψ beteko du. Horrela Fren semantika adierazten da $\bar{x}$-rekin erlazionaturik. Baina zer esan daiteke $F(\bar{a})$ deiari buruz? Bada, $\bar{a}$ parametroek ϕ betetzen badute, F-k $\bar{x}$-rekiko ψ-n betetzen duena izango da, hain zuzen ere, $F(\bar{a})$-ren balioak $\bar{a}$-rekiko duen erlazioa. Har dezagun honako funtzio hau:

```
function GAI (i:integer): real;
begin    {i≠0}
         GAI:= 1/i
end;     {GAI=1/i}
begin    {n≥0}
  s:=0;
  x:=1;
```

$$\texttt{INB} = \left\{ \texttt{s} = \sum_{\texttt{j}=1}^{\texttt{x}-1} 1/\texttt{j} \;\wedge\; 1 \le \texttt{x} \le \texttt{n}+1 \right\}$$

```
  while x≤n do
         begin
                s:=s+GAI(x);
                x:=x+1
         end
  end;
```

$$\left\{ \texttt{s} = \sum_{\texttt{j}=1}^{\texttt{n}} 1/\texttt{j} \right\}$$

Frogatu behar duguna hauxe izango da:

$$\left\{ \texttt{s} = \sum_{\texttt{j}=1}^{\texttt{x}-1} 1/\texttt{j} \;\wedge\; 1 \le \texttt{x} \le \texttt{n} \right\} \; \texttt{s:=s+GAI(x)} \; \left\{ \texttt{s} = \sum_{\texttt{j}=1}^{\texttt{x}} 1/\texttt{j} \;\wedge\; 1 \le \texttt{x} \le \texttt{n} \right\}$$

eta, horretarako, $x \neq 0 \rightarrow GAI(x) = 1/x$ propietatea erabiliko dugu:

1.- $x \neq 0 \rightarrow GAI(x) = 1/x$ (FUNE) (oraindik formalizatu gabea)

2.- $$\left(\left\{ s = \sum_{i=1}^{x-1} 1/j \;\wedge\; 1 \le x \le n+1 \right\} \rightarrow \left\{ s + GAI(x) = \sum_{i=1}^{x} 1/j \;\wedge\; 1 \le x \le n+1 \right\} \right)$$

3.- $$\left\{ s + GAI(x) = \sum_{i=1}^{x} 1/j \;\wedge\; 1 \le x \le n+1 \right\} s := s + GAI(x) \left\{ s = \sum_{i=1}^{x} 1/j \;\wedge\; 1 \le x \le n+1 \right\}$$

Ondorioz, *function* $F\ (\bar{x}:<\text{parametro}-\text{motak}>):<\text{emaitza}-\text{mota}>;\ S;$ badugu eta F-k ez badu albo-ondoriorik, inferentzi erregela hauxe izango da:

$$\frac{\{\phi\}\ S\ \{\psi\}}{\phi^{\bar{a}}_{\bar{x}} \rightarrow \psi^{\bar{a},F(\bar{a})}_{\bar{x},F}}$$

Gogoan izan behar da erregela honetan inferitzen dena frogapenean erabiliko den lehen mailako formula bat dela eta ez zuzentasun partzialeko baieztapen bat. Horren ondorioz beste arazo bat sortzen da. Gatozen argibide gisa funtzio hau ikustera:

```
function f(x:integer): integer;
begin   {true}
        while x≠0 do x:=x;
        f:=x
end;    {f=x ∧ x=0}
```

Ondorioz, edozein f(a) deirentzat, $\text{true} \rightarrow f(a) = a \wedge a = 0$ inferituko dugu edo, nahi bada, f(a)=0, izan ere f(x) indefinitua bait da $x \neq 0$ – rentzat (ez da gelditzen). Benetan egiazkoa hauxe da: $a = 0 \rightarrow f(a) = 0$

Hara zertan datzan arazoa: $\phi^{\bar{a}}_{\bar{x}} \rightarrow \psi^{\bar{a},F(\bar{a})}_{\bar{x},F}$ inferitzeko, $\{\phi\}\ S\ \{\psi\}$ ezezik bukaera ere eskatu behar da. Hori kontuan hartuz, funtzioetarako inferentzi erregela honela geratzen da:

betiere, murriztapen hauek ezarrita:

- ϕ-n eta ψ-n ez dago aldagai lokalik
- ez da albo-ondoriorik gertatzen

4.27. adibidea: *Frogatu ondoko programak, m zenbaki osoa emanda, n balioa kalkulatzen duela, non Fibonacci-ren segidaren n-garren gaia bait da m hori baino hertsiki handiagoa den lehenengo gaia.*

Fibonacci-ren segidaren ondoko definizioa hartuko dugu:

$S_0 = 0,$

$S_1 = 1,$

$S_n = S_{n-2} + S_{n-1}$ *baldin* $n > 1$

```
function fib (i:integer): integer;
var j,a,b: integer;
begin      {i≥0}
  a:=0;
  b:=1;
  for j:=1 to i do          P(j)= {a=s_j ∧ b=s_{j+1}}
          begin
                  b:=a+b;
                  a:=b-a
          end;
  fib:=a
end;       {fib=S_i}
begin             {m≥0}
 n:=1;
 while fib(n) ≤ m do          INB= {s_{n-1} ≤ m ∧ n≥1}
                  n:=n+1;
end;              {s_{n-1} ≤ m ∧ s_n > m}
```

1.– $\{i \geq 0\}$ fib – en gorputza $\{fib = S_i\}$

2.- $(n \geq 0) \rightarrow (fib(n) = S_n)$ 1,(FUNE)

3.- $\{m \geq 0\}$ n:=1 $\{m \geq 0 \wedge n = 1\}$ (AA)

4.- $(m \geq 0 \wedge n = 1) \rightarrow \{S_{n-1} \leq m \wedge n \geq 1\}$

5.- $(S_{n-1} \leq m \wedge n \geq 1 \wedge fib(n) \leq m) \rightarrow (S_{(n+1)-1} \leq m \wedge n+1 \geq 1)$ 2

6.- $\{S_{(n+1)-1} \leq m \wedge n+1 \geq 1\}$ n:=n+1 $\{S_{n-1} \leq m \wedge n \geq 1\}$ (AA)

7.– $\{S_{n-1} \leq m \wedge n \geq 1\}$

while fib(n)≤m do n:=n+1

$\{S_{n-1} \leq m \wedge n \geq 1 \wedge fib(n) > m\}$ 5,6,WHE

8.– $(S_{n-1} \leq m \wedge n \geq 1 \wedge fib(n) > m) \rightarrow (S_{n-1} \leq m \wedge S_n > m)$ 2

9.- $\{m \geq 0\}$

begin n:=1; while fib(n)≤m do n:=n+1 end

$\{S_{n-1} \leq m \wedge S_n > m\}$ 3,4,7,8,(KPE),(ODE)

Ariketak

4.24. Frogatu ondoko programak zenb zenbakiaren faktore guztiak (lehenak nahiz bestelakoak) idazten dituela, 1 eta zenb bera ezik.
HURRENF funtzioaren zuzentasun osoa frogatutzat jo daiteke.

```
var zenb, z, hz: integer;
function   HURRENF (x,f: integer): integer;
  begin    {1 ≤ f ≤ x  ∧  mod f = 0}
  …
  end;     {f < HURRENF ≤ x  ∧  ∀t(f < t < HURRENF → x mod t ≠ 0)
                                  ∧ x mod HURRENF = 0}

begin
  z:=1;
  while z<zenb do
        begin
              hz:=HURRENF (zenb,z);
              write (hz);
              z:=hz
        end
end
```

4.25. Izan bedi E bi dimentsioko osozko array-a, non: E[i, j] > 0 i-garren etapan j-garren postuan sartu zen txirrindulariaren adina den, eta E[i, j] = 0 denean, i-garren etapan j-garrenik ez zela egon adierazten den, j txirrindularik baino gutxiagok bukatu zutela delako etapa, alegia. Ondoko programak postu bat irakurri eta R array-an uzten du etapa bakoitzean postu horretan sartutako txirrindulariaren adina aurretik sartutakoen adinen batezbestekoa baino handiago ('+') ala txikiagoa den ('-'). Irakurritako postuan txirrindularirik sartu ez dela adierazteko 'N' balioa erabiliko da. Dokumenta ezazu programa hau tarteko asertzio eta inbarianteekin.

```
cons
  ek = ? ;              (* etapa-kopurua *)
  tk = ? ;              (* hasi diren txirrindularien kopurua *)
var
  E : array [1..ek, 1..tk] of integer;
  R : array [1..ek] of char;
  p, i, k : integer;

function BA (N : array [1..ek, 1..tk]; etapa, postu : integer) : real;
  var       batura, j : integer;
  begin  {
            batura := 0;
            for j := 1 to postu-1 do  batura := batura + N[etapa, j];   P(j) ={ }
            BA := batura / (postu - 1);
  end;       {

begin       {
  read (p); i := 1;
  while (E[i, p] ≠ 0 and i ≤ne) do      INB  =  {  }
            begin
                      if BA (E, i, p) ≤ E[i, p] then R[i] := '+' else R[i] := '-';
                      {
                      i := i + 1;
            end;
  if i < ek then
                      for k := i to ek do R[k] := 'N';  P(k) = { }
end.        {
```

5. PROGRAMA-ERATORPEN FORMALA

5.1. METODOAREN FILOSOFIA, INTERESGARRITASUNA ETA MUGAK

Programa baten espezifikazio batekiko zuzentasunaren frogapena, diseinatutakoan erabilitako arrazonamenduaren formalizazioa da; nolabait, programa horren diseinuan erabili diren ideiak, propietateak eta arrazonamenduak dira garrantzizkoenak zuzentasuna frogatzerakoan. Ikuspuntu honetan oinarrituta berehalakoan bururatzen zaigu ondoko ideia:

"Programa eta zuzentasunaren frogapena aldi berean egin behar dira, frogapenak programaren eraiketa gidatzen duela".

Programak ideia berri honi jarraikiz diseinatuz gero, zuzentasunaren frogapen-eskemez gain, programagintzan erabilitako arrazonamenduaren trazak ere lortzen dira eta, bata nahiz bestea, biak dira oso erabilgarriak mantenimendu-fasean. Gogoan izan honakoa: "Programa behin egiten da eta askotan irakurri".

Nolanahi ere, formalismoari gehiegi atxekitzea metodoaren muga nabaria izan daiteke, eta horregatixe, ez da beharrezkoa, ez eta desiragarria ere, formalismoari gehiegi lotzea.

Intuizioak eta zentzu komunak (bestelako tresneriarik gabe) diseinu txarrak eta oker gehiegi sor ditzakete, baina formalismo zurrunegiak ere nahasteak dakartza, xehetasunetan gehiegi murgiltzearen ondorioz. Egokiena, beraz, formalismo eta intuizioaren arteko oreka lortzea da, axolagabeko xehetasun formalak albo batera utziaz, diseinu fidagarria eta dokumentazio aproposa ihardestea errazten duena.

Delako oreka hau ohitura bilakatu da matematiketan: frogapenak irakurlea konbentzitu egin behar du, baina horretarako ulerterraza eta ondo eratutakoa izan behar du, punturik azpimarragarrienak eta zailtasun handienekoak agerian utziz. Gisa berean, oreka honek ohiturazko izan behar du programatzailearentzat ere diseinatu eta dokumentatzean.

Hauxe da filosofia: "Erabili intuizioa egokiera dagoenean eta besterik gabe nahikoa denean, eta teoria euskarri bezala zailtasunak daudenean".

Horretarako Gries-ek hiru betebehar nabarmentzen ditu:

-intuizioa edukitzea,

-teoriarekin erraztasuna edukitzea,

-biak orekatzen jakitea.

Horietatik gutxien menperatzen dena teoriarekin erraztasuna edukitzearena izan ohi da, eta bereziki hori lantzera joko dugu atal honetan.

Hala ere, aurrera jo baino lehen komeni da zenbait xehetasun azaltzea:

- Metodo honen ezagutza oinarri aproposa da edozein programatzailerentzat, baina, bistakoa denez, ezin daiteke erabili edonolako problema-motatan. Badira bestelako metodo eta kontzeptuak ezagutu eta, problemak hala eskatuz gero, erabili behar direnak.
- Metodoaren helburuetako bat programen diseinu-fasean erroreak ekiditea da. Metodo hau erabiltzeak, nolanahi ere, ez du ziurtatzen erroreak sortuko ez direnik, baina sortzeko arriskua gutxiagotu egiten du, hori bai.
- Horrelako metodo formalaz programatzeak zailagoa dela ematen du. Baina, kontuz, intuizio hutsez programatzea zailtasunak enoratzea besterik ez da, geroago ageriko diren eta konplexuago gertatuko diren zailtasunen aurrean "itsuarena egitea", alegia.

Programatzeko era formalago honek hauxe du oinarrizko printzipio: programazioa, ondoko baldintzak adierazitakoa betetzeko helburua duen iharduera da. Hau da, $\{\phi\}$ [P] $\{\psi\}$ betetzen duen P programa diseinatzea, metodo honen ikuspuntutik, ψ ondoko baldintzak finkatzen duena betetzen duen programa diseinatzea besterik ez da.

ψ arretaz aztertu eta ulertzeak, eta intuizioaz eta zolitasunaz baliatzeak, programaren diseinurako ideiak ematen dizkigute. Ideia hauek gauzatzean ϕ aurreko baldintzak emaitzaren betebeharrak bermatu behar dituela kontuan hartu behar da.

Hortik dator, hain zuzen ere, "aurreko baldintza ahulena" izeneko nozioa, azken batean, gutxienezko betebeharra adierazten duena.

Programa bat eratortzeko ezinbestekoa da espezifikazio zehatz eta guztiz ulergarri batetik abiatzea. Metodoaren filosofiak berak argi uzten du problema sakonetik ezagutu behar dela soluzioetara jotzen hasi aurretik.

5.2. WP PREDIKATU-TRANSFORMATZAILEA

Edozein I agindu eta ψ ondoko baldintzarentzat, I eta ψ-ren aurreko baldintza ahulen deritzon wp(I,ψ) formula definituko dugu.

(I,ψ) formulak errepresentatzen duen egoera-multzoko, eta soilik multzo horretako edozein egoeratatik abiatuta, Iren exekuzioa denboraldi mugatuan ψ betetzen duen egoera batean amaitzen da.

wp predikatu-transformatzailea funtzio gisa ere har daiteke:

wp (I,-) : Lehen mailako formulak $\rightarrow$ Lehen mailako formulak

$\psi \rightarrow$ wp(I,ψ)

non $\{wp\ (I, \psi)\}\ [I]\ \{\psi\}$ baieztapen zuzena bait da eta gainera:

$\{\phi\}\ [I]\ \{\psi\} \Leftrightarrow \phi \rightarrow wp\ (I, \psi)$

Askotan, aurreko baldintza ahulena baino beste ϕ gogorragoa gehiago interesatzen zaigu, beraz ϕ erregelak wp(I,ψ)-k errepresentatzen dituen egoeren azpimultzoa adieraziko du. Kasu honetan, $\phi \rightarrow wp\ (I, \psi)$ frogatu behar da.

Orain ikustera goazen wp predikatu-transformatzailearen definizioa programazio--lengoaiaren (gure kasuan Pascal-aren) semantika finkatzeko beste modu bat baizik ez da, Hoare-ren sistema formalaren oso antzekoa (areago, sistema honetan oinarritutakoa).

Aldagai sinpleei eginiko asignazioa

ψ formularen eta aldagai sinple bati eginiko asignazioaren aurreko baldintza ahulena hauxe da:

$$wp(x:=t, \psi) = def\ (t) \wedge \psi_x^t$$

non *def(t)*-k t definituta dagoen egoera guztien multzoa errepresentatzen bait du.

Array-en osagaiei eginiko asignazioa

$$wp(A[i]:=t, \psi) = heinean\ (i) \wedge def\ (t) \wedge \psi_A^{A,t//i}$$

non *heinean (i)*-k, i aldagaiak A array-aren mugen arteko balioren bat dueneko egoeren multzoa errepresentatzen bait du eta *def(t)*-k lehengo esanahi berbera bait dauka.

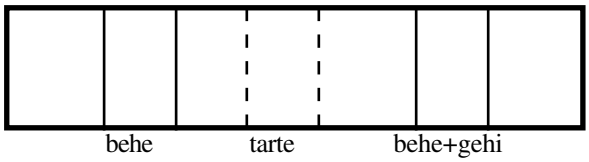
Sekuentzi konposaketa

$$wp\ (I_1; I_2, \psi) = wp(I_1, wp(I_2, \psi)),$$

non $wp(I_2,\psi)$ tarteko asertzio ahulena bait da.

Ikus dezagun adibide batez programa-zati sinple baten eratorpena.

5.1. adibidea: *Suposa dezagun A array-a partizioka lantzen ari garela eta behe, tarte eta gehi aldagaiak ditugula, non behe≤tarte<behe+gehi bait da.*



Jo dezagun *behe* eta *behe+gehi* adierazpenen tartean dagoen *tarte* indizeko elementua landu dela eta, ondorioz, *behe* eguneratu nahi dugula. *Behe*-ren balio berria *tarte+1* izatea nahi dugu, baina bestalde ez dugu *behe+gehi* muga aldatzerik nahi. Partizio eguneratua A[tarte+1..behe+gehi] zabalera duena izango da; baina, horra koxka, zein izango da *gehi*-ri esleitu behar zaion balioa *behe+gehi* adierazpena hasierako horretan manten dadin?

Izan bedi b behe+gehi adierazpenaren hasierako balioa, hau da, mantendu nahi dugun balioa. Nabarmena da x-ren balioa jakin nahi dugula ondoko hirukotean:

{behe+gehi=b} behe:=tarte+1; gehi:=x {behe+gehi=b}

Honela kalkula daiteke:

1) wp(behe:=tarte+1; gehi:=x, behe+gehi=b) = ϕ

2) ϕ-k x eta b-ren arteko erlazioa ematen digu, hau da, behe+gehi adierazpenaren hasierako balioa b dela jakinda erraz lor dezakegu x;

beraz:

1) wp (behe: = tarte +1; gehi: =x; behe + gehi = b) = (tarte + 1+ x = b)

2) $(\text{tarte} + 1 + x = b \ \wedge \ b = \text{behe} + \text{gehi}) \rightarrow (\text{tarte} + 1 + x = \text{behe} + \text{gehi}) \rightarrow$
$(x = \text{behe} + \text{gehi} - \text{tarte} - 1)$

Hortaz, gehi-ri asignatu beharreko balioa behe+gehi-tarte-1 da, eta agindu konposatua honela osatuko da:

<u>begin</u> behe:=tarte+1; gehi:=behe+gehi-tarte-1 <u>end</u>

Bestalde, eratorritako agindua zuzena dela frogatzeko, $\{\phi\}\ [I]\ \{\psi\}$ betetzen dela alegia, hauxe da bidea:

1) wp (P, ψ) kalklatu, eta

2) $\phi \rightarrow$ wp (I, ψ) egiazkoa dela frogatu.

Erabil dezagun frogapen-modu hau adibide honetan:

5.2. adibidea: *Izan bedi {true} [x:=a+b; x:=x/c; x:=x∗x] {x=(a+b/c)^2} frogatu nahi duguna:*

1)

$$wp\left(x:=a+b;\ x:=x/c;\ x:=x*x\ ,x=(a+b/c)^2\right)=$$

$$wp\left(x:=a+b;\ x:=x/c,\ wp\left(x:=x*x,\ x=(a+b/c)^2\right)\right)=$$

$$wp\left(x:=a+b;\ x:=x/c,\ x*x=(a+b/c)^2\right)=$$

$$wp\left(x:=a+b,\ wp\left(x:=x/c,\ x*x=(a+b/c)^2\right)\right)=$$

$$wp\left(x:=a+b,\ c\neq 0\ \wedge\ x/c*x/c=(a+b/c)^2\right)=$$

$$c\neq 0\ \wedge\ (a+b/c)*(a+b/c)=(a+b/c)^2=$$

$$c\neq 0$$

2) true $\rightarrow$ c $\neq$ 0 formularen ebaluaketak faltsu balioa ematen du, beraz hasierako baieztapena ez da zuzena.

Hau da, agindua zuzena izan dadin aurreko baldintzak bete beharko lituzkeen gutxienezko baldintzak c $\neq$ 0 dira, eta, bistan denez, ez ditu betetzen.

Baldintzazko agindua

If-then-else erako aginduen aurreko baldintza ahulena honela definitzen da:

$$wp\ (\underline{\text{if}}\ B\ \underline{\text{then}}\ I_1\ \underline{\text{else}}\ I_2\ ,\ \psi)\ =\ def\ (B)\wedge(B\rightarrow wp(I_1,\psi))\wedge(\neg B\rightarrow wp(I_2,\psi))$$

Programa bat egiten ari garenean, baliteke eraiki behar dugun baldintzazko sententziaren aurreko baldintza (aurreko aginduaren ondoko baldintza izango dena) aldez aurretik ezagutzea. Kasu honetan egokia da honako teorema hau erabiltzea:

Teorema: ϕ aurreko baldintza ezaguna bada, eta ondoko propietateak betetzen baditu:

a) $\phi\rightarrow def\ (B)$,

b) $\phi\wedge B\rightarrow wp(I_1,\psi)$

c) $\phi\wedge\neg B\rightarrow wp(I_2,\psi)$

Orduan: $\phi\ \rightarrow$ wp (if B then I_1 else I_2, ψ) formula ere egiazkoa da.

Frogapena:

b) eta c)-ren ondorioz:

$$\left(\phi \wedge B \rightarrow wp\left(I_1, \psi\right)\right) \wedge \left(\phi \wedge \neg B \rightarrow wp\left(I_2, \psi\right)\right) =$$
$$\left(\neg\left(\phi \wedge B\right) \vee wp\left(I_1, \psi\right)\right) \wedge \left(\neg\left(\phi \wedge \neg B\right) \vee wp\left(I_2, \psi\right)\right) =$$
$$\left(\neg\phi \vee \neg B \vee wp\left(I_1, \psi\right)\right) \wedge \left(\neg\phi \vee B \vee wp\left(I_2, \psi\right)\right) =$$
$$\neg\phi \vee \left(\left(\neg B \vee wp\left(I_1, \psi\right)\right) \wedge \left(B \vee wp\left(I_2, \psi\right)\right)\right) =$$
$$\phi \rightarrow \left(\left(B \rightarrow wp\left(I_1, \psi\right)\right) \wedge \left(\neg B \rightarrow wp\left(I_2, \psi\right)\right)\right)$$

eta a) kontuan hartuz gero:

$$\phi \rightarrow \left(def\left(B\right) \wedge \left(B \rightarrow wp\left(I_1, \psi\right)\right) \wedge \left(\neg B \rightarrow wp\left(I_2, \psi\right)\right)\right)$$

Egindako frogapen horretatik baldintzazko aginduak garatzeko bidea ondoriozta daiteke. Bide horretako urratsak dira honako hauek:

1- I_1 agindu bat aurkitu, delako aginduaren exekuzioak ψ ondoko baldintza betetzen duela.

2- wp erabiliz aurkitu zein kasu den; hau da, B adierazpen boolearra aurkitu, non
$$\phi \rightarrow def\left(B\right) \quad \text{eta} \quad \phi \wedge B \rightarrow wp\left(I_1, \psi\right)$$

3- B eta, beraz, $\neg B$ ezagututa, eratorri I_2, non
$$\phi \wedge \neg B \rightarrow wp\left(I_2, \psi\right)$$

5.3. adibidea: *Demagun x balioaren bilaketa bitarra egiten ari garela A[1..n] array ordenatuan.*

Jo dezagun hau dela aurreko baldintza:

$$\phi = \left(\text{ordenatua}\left(A\left[1 \ldots n\right]\right) \wedge x \in A\left[i..j\right] \wedge \ 1 \le i < k < j \le n\right),$$

Bilaketa burutzeko [i..j] tartea estutu egin behar da, baina hori bai, $x \in A\left[i..j\right]$ propietateari eutsiz.

1) *i:=k* agindua hartzen badugu:

wp(i:=k, $x \in A[i..j]$) = $x \in A[k..j]$

2) $\phi \wedge A\left[k\right] \le x \rightarrow x \in A\left[k..j\right]$ eta $\phi \rightarrow def\left(A\left[k\right] \le x\right)$

Beraz, eratorriko duguna: <u>if</u> A[k]≤x <u>then</u> i:=k

3) $\neg B = A\left[k\right] > x$ denez,

$$wp\left(j := k, \ x \in A\left[i..j\right]\right) = x \in A\left[i..k\right] \quad \text{eta} \quad \phi \wedge A\left[k\right] > x \rightarrow x \in A\left[i..k\right]$$

Honenbestez, bada, agindu hau lortu dugu:

<u>if</u> A[k]≤x <u>then</u> i:=k <u>else</u> j:=k

baina agindu hori ezezik bere zeregina zuzen bete dezan gogoan izan ditugun propietate eta ideien traza ere.

Ariketak

5.1. Asmatu wp (if B then I,ψ) formularako definizio bat.

5.2. Erabaki ondoko baieztapenak egiazkoak ala faltsuak diren, erantzunak behar bezala frogatuz:

a) P(x) ν P(y) $\rightarrow$ wp(if not P(x) then x:=y, P(x))

b) x=a $\wedge$ y=0 $\rightarrow$ wp(begin x:=x/z; y := y+1 end , x = a/z^y)

c) 0<x$\leq$z $\rightarrow$ wp(if x<z then r:=x else r:=0, r = x mod z)

d) P(y) $\rightarrow$ wp(if (P(x) and P(y)) then z:=x else z:=y, P(z))

e) $(z=(x+1)^2 \wedge y=2*x+1) \rightarrow$ wp(x:=x+1; y:=y+2; z:=z+y, $z=(x+1)^2$)

f) (x=1 $\wedge$ y=2) $\rightarrow$ wp (x:=2$*$(x+y); y:=x+y; x:=(x+2)/y; y:=y-6, x=1 $\wedge$ y=2)

g) (x>0) $\rightarrow$ wp (if x>y then abs:=-y else abs:=y, abs=$|$y$|$)

h) (x=a $\wedge$ y<x) $\rightarrow$ wp (if x<0 then y:=-y, y>a $\wedge$ x=a)

i) (x=0) $\rightarrow$ wp (x:=2; y:=4/x, y=2)

Iterazioak

wp(<u>while</u> B <u>do</u> I, ψ) formulak errepresentatzen duen egoera-multzoko eta soilik multzo horretako edozein egoeratatik abiatuta, Iren exekuzioak iterazio-kopuru finituan ψ betetzen den egoera bat sortzen du (eta, horrezaz gain, $\neg$ B ere beteko da).

Aurreko baldintza horren esanahia formalizatzeko lagungarri gertatuko zaigu, *<u>while</u> B <u>do</u> I* agindua emanda, $H_k(\psi)$ formula definitzea:

$H_k(\psi)$ formulak errepresentatzen duen egoera-multzoko, eta soilik multzo horretako, edozein egoeratatik abiatuta, Iren exekuzioa gehienez ere k iteraziotan amaitzen da, eta amaitutakoan ψ (eta $\neg$ B ere) betetzen den egoera bat gertatzen da.

Ikus dezagun zer nolako forma duten $H_k(\psi)$ delakoek:

$\left(H_0(\psi) = \psi \wedge \neg B\right)$= Iterazioaren gorputza 0 aldiz exekutatzen deneko egoera guztien multzoa.

$H_1(\psi) = (\psi \wedge \neg B) \wedge \left(wp(I, \psi \wedge \neg B) \wedge B\right)$

= Iterazioaren aurretiko egoeren multzoa, non I 0 edo 1 aldiz exekutatzen den.

$= H_0(\psi) \vee \left(B \wedge wp\ (I, H_0(\psi))\right)$

Orokorrean:

$H_k(\psi) = H_0(\psi) \vee \left(B \wedge wp\ (I, H_{k-1}(\psi))\right)$, k>1 denean.

Ondorioz, <u>while</u> aginduaren eta ψ ondoko baldintzaren aurreko baldintza ahulena honela defini daiteke:

$$wp(\underline{\text{while}}\ B\ \underline{\text{do}}\ I, \psi) = \exists k (k \geq 0 \wedge H_K(\psi))$$

5.4. adibidea: *wp(<u>while</u> i<n <u>do</u> <u>begin</u> i:=i+1; F:=F*i <u>end</u>, F=n!) kalkulatu.*

$H_0(F = n!) = (F = n! \wedge i \geq n)$

$H_1(F = n!) = (F = n! \wedge i \geq n) \vee \left(i < n \wedge wp(i := i+1;\ F = F*i,\ F = n! \wedge i \geq n)\right) =$

$= (F = n! \wedge i \geq n) \vee (F = (n-1)! \wedge i = n-1)$

$H_2(F = n!) = (F = n! \wedge i \geq n) \vee (F(n-1)! \wedge i = n-1) \vee (F(n-2)! \wedge i = n-2)$

Hedapenez:

$H_k(F = n!) = (F = n! \wedge i \geq n) \vee (F(n-1)! \wedge i = n-1) \vee \ldots \vee (F(n-k)! \wedge i = n-k)$

$= (F = n! \wedge i \geq n) \vee (F = i! \wedge n-k \leq i < n)$

Beraz:

wp(<u>while</u> i<n <u>do</u> <u>begin</u> i:=i+1; F:=F*i <u>end</u>, F=n!) =

$(F = n! \wedge i \geq n) \vee \exists k (k > 0 \wedge F = i! \wedge n-k \leq i < n)$

5.5. adibidea: *Aurkitu ondoko programa-zatiaren aurreko baldintza ahulena:*

```
while r≥y do
 begin
   r:=r-y;  q:=q+1
 end
```

$\psi = \{x = q*y + r \ \wedge \ 0 \le r < y\}$

$H_0(\psi) = x = q*y + r \ \wedge \ 0 \le r < y$

$H_1(\psi) = H_0(\psi) \ \vee \ (x = q*y + r \ \wedge \ y \le r < 2*y)$

Orokorrean:

$H_k(\psi) = H_{k-1}(\psi) \ \vee \ (x = q*y + r \ \wedge \ k*y \le r < (k+1)*y)$

Beraz:

$wp(\underline{\text{while}}\ r \ge y\ \underline{\text{do}}\ \underline{\text{begin}}\ r := r - y;\ q := q + 1\ \underline{\text{end}},\ x = q*y + r \ \wedge \ 0 \le r < y) =$

$\exists k(k \ge 0 \ \wedge \ x = q*y + r \ \wedge \ k*y \le r < (k+1)*y)$

5.6. adibidea: *Har dezagun orain programa osoa:*

```
begin
r:=x; q:=0;
while r≥y do
 begin
   r:=r-y;  q:=q+1
 end
end.
```

$\psi = \{x = q*y + r \ \wedge \ 0 \le r < y\}$

Programa osoa eta inbariantea ditugula galdera hau bururatzen da: zein da programa honen aurreko baldintza ahulena?

Ea bada, kalkula dezagun:

$wp(r := x;\ q := 0,\ \exists k(k \ge 0 \ \wedge \ x = q*y + r \ \wedge \ k*y \le r < (k+1)*y)) =$

$\exists k(k \ge 0 \ \wedge \ k*y \le x < (k+1)*y) =$

$x \ge 0 \ \wedge \ y > 0$

Hala ere, orain artean ikusitako while-ren definizio formala ez da oso erabilgarria, eta ez du laguntza askorik eskaintzen iterazioen diseinurako. Litekeena da aurreko baldintza bat egotea, ez derrigorrez ahulena, baina iterazioak lantzeko lagungarri dena; hain zuzen, inbariantea bera. Inbarianteari borne-adierazpena eransten badiogu, horra iterazioak eraikitzeko behar duguna, nahikoa eta zalantzarik gabe interesgarriena.

Iterazioei buruz ikusitakoa ondorengo teoremetan laburbil daiteke:

Inbariantearen teorema:

while B do I agindua emanda, baldin eta $INB \wedge B \rightarrow wp(I, INB)$ betetzen duen INB inbarianterik badago, orduan

$$INB \wedge wp(\text{while } B \text{ do } I, true) \rightarrow wp(\text{while } B \text{ do } I,\ INB \wedge \neg B)$$

Bukaeraren teorema:

INB, *while B do I* aginduaren inbariantea bada eta e adierazpen oso bat, non:

1) $INB \wedge B \rightarrow e > 0$
2) $INB \wedge B \wedge e = e_0 \rightarrow wp(I, e < e_0)$

orduan INB→wp (while B do I, true).

Eta bi teorema hauek batuz:

Teorema:

while B do I agindua emanda, baldin INB eta e badaude non:

1) $INB \wedge B \rightarrow wp(I, INB)$
2) $INB \wedge B \rightarrow e > 0$
3) $INB \wedge B \wedge e = e_0 \rightarrow wp(I, e < e_0)$

orduan $INB \rightarrow wp(\text{while } B \text{ do } I,\ INB \wedge \neg B)$

Ideia hauek erabili nahi ditugu iterazioak diseinatzeko. Eman dezagun ϕ aurreko baldintzaz eta ψ ondoko baldintzaz adierazitako espezifikazioa betetzen duen iterazioa eta dagokion hasieraketa diseinatu nahi ditugula, eman beharreko pausoak hauek izango dira:

1) INB inbariantea planteatu.
2) e adierazpena asmatu.
3) INB eta e adierazpenetik abiatuta iterazioa eta dagokion hasieraketa eratorri.

Egiaztapenari buruzko gaian jadanik diseinatutako iteraziozko aginduen inbariante eta borne-adierazpenen asmaketa landu da. Oraingoan, berriz, iterazioa ezagutu aurretik idatzi nahi dira inbariante eta borne-adierazpenak, diseinuaren gidari izan daitezen.

Nola lortu inbarianteak

Oinarrizko ideia hauxe da: Inbariantea ondoko baldintza baino formula ahulagoa da eta aurreko baldintza bere baitan biltzen du. Gainera, iterazioaren edozein pausoren hasieran konputazio-egoera zein den adierazten duen formula da (iterazioaren semantika).

Ondoko baldintza ezagutuz gero, hori ahulduta inbariantea lor daiteke, pausoeroko egoera posible guztiak errepresentatzen dituela, bereziki hasierakoak. Bada, beraz, apimarratu beharreko aspektu bat, formulak ahultzearena.

Ondoko baldintza ezagututa inbariantea gorpuzteko erabilgarri gerta daitezkeen formulak ahultzeko modu batzuk ikusiko ditugu.

a) Konjuntzio bat kentzea:

Baldin ψ bi formulen konjuntzio bada $\psi = \psi_1 \wedge \psi_2$, orduan ondoko baldintza baino ahulagoa den INB inbariantea INB= ψ_1 izan daiteke. Kasu honetan ψ_2-k $INB \wedge \neg B$ formularen ondorioa izan beharko du, non B hori iterazioaren baldintza boolearra den. Ikus ditzagun adibide batzuk:

5.7. adibidea: *Erro karraturako hurbilketa*

$$\psi = 0 \le a^2 \le n < (a+1)^2$$
$$INB = 0 \le a^2 \le n$$

5.8. adibidea: *Array batean bilaketa lineala.*

$$\psi = 1 \le i \le n \ \wedge x \notin A[1..i-1] \wedge x = A[i]$$
$$INB = 1 \le i \le n \ \wedge x \notin A[1..i-1]$$

b) Konstante bat aldagai batez ordeztu. Aldagai horrek har ditzakeen balioen artean egon behar du konstantearen balioak ere. Adibideak:

5.9. adibidea: *Array baten elementuen batura.*

$$\psi = \left(s = \sum_{j=1}^{i} A[j] \right)$$
$$INB = \left(s = \sum_{j=1}^{i} A[j] \ \wedge \ 1 \le i \le n \right)$$

5.10. adibidea: *Array baten maximoa aurkitu.*

$$\psi = \left(x = \max\left(B[n_1..n_2] \right) \right)$$
$$INB = \left(x = \max\left(B[n_1..i] \right) \ \wedge \ n_1 \le i \le n_2 \right)$$

c) Aldagai bat beste batez ordeztu. Azken aldagai honen eremuan egon behar du aurreko aldagaiaren balioak ere.

5.11. adibidea: *Erabaki zenbaki bat lehena den ala ez:*

$$\psi = \left(b = \text{true} \leftrightarrow \forall j\left(2 \le j \le x - 1 \rightarrow x \ \underline{\text{mod}}\ j \neq 0\right)\right)$$

$$\text{INB} = \left(b = \text{true} \leftrightarrow \forall j\left(2 \le j < d \rightarrow \underline{\text{mod}}\ j \neq 0\right) \wedge 2 \le d \le x\right)$$

5.12.adibidea: *Erro karraturako hurbilketa:*

$$\psi = \left(0 \le a^2 \le n < (a+1)^2\right)$$

$$\text{INB} = \left(0 \le a^2 \le n < b^2 \ \wedge \ a < b\right)$$

d) Ikusi ditugun moduak kasu askotan erabil daitezke, baina badaude inbariantea ahultzeko beste era batzuk, ez hain arruntak, baina zenbaitetan egokiak direnak. Esate baterako, disjuntzio bat gehitzea:
Ondoko baldintza ψ bada, inbariante gisa $\text{INB} = \psi \vee \gamma$ hartzea, non γ formularen bat bait da.

e) Batzuetan era desberdinen konbinazioa erabili behar da.

f) Orain arte ondoko baldintza bakarrik hartu dugu kontuan, baina batzuetan aurreko baldintza ezagutzea ere beharrezkoa da (batez ere honek hasierako balioak finkatzen dituenean). Kasu hauetan aurreko eta ondoko baldintzen konbinazioa ahuldu behar da.

5.13. adibidea*:*

$$\phi = \forall i\left(1 \le i \le n \rightarrow A[i] = a_i\right)$$

$$\psi = \forall i\left(1 \le i \le n \rightarrow A[i] = a_i + p\right)$$

$$\text{INB} = \forall i\left(1 \le i \le k \rightarrow A[i] = a_i + p\right) \ \wedge \ \forall i\left(k < i \le n \rightarrow A[i] = a_i\right)$$

Nola lortu borne-adierazpenak

Lehenxeago ikusi denez, borne-adierazpenak batzuetan iterazio bat amaitu dadin falta diren urratsen kopurua mugatzen du eta, beste batzuetan, soilik programaren bukaera frogatzeko balio du.

Sarritan inbarianteak berak erakusten du zein den bigiztaren borne-adierazpena, eta iterazioa diseinatzeko dagoenean ere baieztapen hau zuzena da. Ikus ditzagun adibideok:

5.14. adibidea: *Zein da 5.9. adibideko iterazioaren borne-adierazpena?*

Intuizioz edo, asma daiteke borne-adierazpena oraindik batu gabeko elementuen kopurua dela, hau da, *n-i.*

5.15 adibidea: *Zein da 5.11 adibideko iterazioaren borne-adierazpena?*

Kasu horretan pentsa daiteke landu gabeko zatitzaile posibleen kopurua izango dela borne-adierazpena, hau da, *x-d*. Azpimarratzekoa da adierazpen horrek kasurik txarreneko pausu-kopurua bornatzen duela.

5.16. adibidea: *Zein da 5.12 adibideko iterazioaren borne-adierazpena?*

Bada, oraingoan ere intuizioz, borne-adierazpen gisa aztertu behar den tartearen luzera hartuko dugu: *b-a+1*

Nola eratorri iterazioak inbariante eta borne-adierazpenetatik

Jo dezagun iterazio bat eta dagokion hasieraketa diseinatu nahi ditugula eta iterazioak betebehar hauek dituela:

a) ϕ eta ψ formulek emandako espezifikazioa bete,

b) INB inbariantea kontserbatu, eta

c) Bestalde, e adierazpenak pauso-kopurua bornatu behar du.

Baldintzok ezarritakoaren pean, ondoko urratsok eman beharko dira iterazioen eratorpen formalean:

1.- HAS hasieraketa eman eta zuzena dela frogatu:

$\phi \rightarrow$ wp(HAS, INB)

2.- B baldintza boolearra asmatu, non

$INB \wedge \neg B \rightarrow \psi$ bete behar duen.

3.- Iterazioaren gorputza eraiki, e adierazpena beherarazi eta inbariantea kontserbatzen duela.

5.17. adibidea: *A[1..m,1..n] bi dimentsioko array-ean x elementuaren bilaketa burutzen duen programa formalki erator ezazu.*

Espezifikazioa, ondoko formulek emandakoa:

$\phi = (n \geq 1 \wedge m \geq 1)$

$\psi = (1 \leq i \leq m \ \wedge \ 1 \leq j \leq n \ \wedge \ x = A[i,j]) \vee (i = m+1 \wedge x \notin A[1..m,1..n])$

1.- Eman dezagun ondokoa hartzen dugula diseinatuko dugun iterazioaren inbariantetzat: "Array-a errenkadaka aztertu dugu goitik behera, eta errendaka bakoitza ezkerretik eskuinera, orain gauden A[i,j] elementuraino".

Hau da:

$INB = 1 \leq i \leq m+1 \ \wedge \ 1 \leq j \leq n \ \wedge \ x \notin A[1..(i-1),1..n] \wedge x \notin A[i,1..(j-1)]$

2.- Borne-adierazpena aztertu gabeko osagaien kopurua izango da:

e = n∗(m-i+1)-j+1

3.1.- Hasieraketa eman eta zuzentasuna frogatu:

i:=1; j:=1

wp(i:=1; j:=1, INB) = $n \geq 1 \wedge m \geq 1$, eta baldintza hau ϕ da, hain zuzen ere.

3.2.- Aurkitu B baldintza boolearra, $INB \wedge \neg B \rightarrow \psi$ beteko dela ziurtatuz.

Nahikoa da horretarako: $\neg B = \left(i \leq m \wedge x = A[i,j]\right) \vee (i = m+1)$

Eta $\neg B$ ezagututa B honela laburtu daiteke:

$$B = \neg\left(i \leq m \wedge x = A[i,j]\right) \wedge i \neq m+1 = \left(i > m \vee x \neq A[i,j]\right) \wedge i \neq m+1 =$$
$$= \left(i = m+1 \vee x \neq A[i,j]\right) \wedge i \neq m+1 = \left(x \neq A[i,j] \wedge i \neq m+1\right)$$

Oraingoz, programa hau eratorri dugu:

```
begin {true}
        i:=1; j:+1;  {INB}
        while i≠m+1 ∧ x≠A[i,j] do aurreratu (i,j)
end     {ψ}
```

3.3.- Iterazioaren gorputza eraiki, kasu honetan *aurreratu (i,j)*.

Borne adierazpena $e = n*(m-i+1)-j+1$ da eta *aurreratu(i,j)* ekintza soilik $INB \wedge B$ betetzen bada exekutatzen da, hau da, $i \leq m \wedge j \leq n \wedge x \neq A[i,j]$ gertatzen denean. Beraz, ekintza honek ondokoa bete beharko du:

$\{1 \leq i \leq m \wedge 1 \leq j \leq n \wedge x \neq A[i,j]\}$ *aurreratu(i,j)* $\{1 \leq i \leq m+1 \wedge 1 \leq j \leq n\}$

Soilik baldin j<n bada j-ri bat gehitzen zaio. j=n bada, errenkada bukatu denaren seinale, eta hurrengora pasa beharko dugu, hau da: i:=i+1; j:=1.

Hauxe lortuko dugu:

```
        {1≤i≤m ∧ 1≤j≤n ∧ x≠A[i,j]}
if j<n then   {j<n ∧ 1≤i≤m}
        j:=j+1
      else
        begin  {1≤i≤m ∧ j=n}
        i:=i+1 ; j:=1
        end
      {1≤i≤m+1 ∧ 1≤j≤n}
```

Honenbestez, programa dokumentatua:

```
begin {true}
  i:=1; j:=1:
{1≤i≤m+1 ∧ 1≤j≤n ∧ x∉A[1..(i-1),1..n] ∧ x∉A[i,1..(j-1)]}
        {e=n*(m-i+1)-j+1}
  while (i≠m+1) and (x≠A[i,j]) do
        {1≤i≤m ∧ 1≤j≤n ∧ x∉A[1..(i-1),1..n] ∧
         x∉A[i,1..(j-1)] ∧ x≠A[i,j]}
  if j<n then   {1≤j<n ∧ x∉A[1..(i-1),1..n] ∧ x∉A[1..i,1..n]}
        j:=j+1
      else begin   {1≤i≤m ∧j=n x∉A[1..i,1..n]}
             i:=i+1; j:=1
           end
end.
                {(1≤i≤m ∧ 1≤j≤n ∧ x=A[i,j]) ∨
                  (i=m+1 ∧ x∉A[1..m,1..n])}
```

Ilun xamar geratzen den aspektu bakarra e adierazpenaren beherapena da, kasu honetan:

```
begin   {1≤i≤m ∧ j=n}
        i:=i+1; j:=1
end
```

Ikus dezagun wp erabiliz benetan e-ren balioa txikiagotu egiten dela:

wp(i:=i+1; j:=1, n∗(m-i+1)-j+1<b) = n∗(m-(i+1)+1)-1+1<b = n∗(m-i)<b

eta, beraz,

$$1 \le i \le m \wedge j = n \wedge n*(m-i+1)-j+1 = b \rightarrow n*(m-i+1)-n+1 = b \rightarrow$$
$$n*(m-i)+1 = b \rightarrow n*(m-i) < b$$

Lortutako programak arazo bat dauka while-ren baldintza balioztatzean, Pascal-eko adierazpenak balioztatzeko moduagatik. Arazo hau ekiditeko aldaketaren bat egin beharko litzateke lortutako programan:

```
begin
  i:=1;  j:=1;
  aurkitua:= false;
  while (i<>m+1) and (not aurkitua) do
        if x= A[i,j] then aurkitua:= true
                     else aurreratu (i,j)
end.
```

5.18 adibidea: *n zenbaki arrunta emanda, erro karratura behetik gehien hurbiltzen zaion zenbaki osoa kalkulatu (suposatuz ez dugula ez sqrt, ez trunc bezalako eragiketarik).*

Espezifikazioa:

$\phi = n \geq 0$

$\psi = a^2 \leq n < (a+1)^2$

Hasierako ideia 'a' aldagaia ahal deneraino handiagotzen duen iterazioa diseinatzea da. Ideia honek eraginda:

$INB = a^2 \leq n$

$e = n - a^2$

$B = (a+1)^2 \leq n$, non $INB \wedge \neg B = \psi$

Inbariantea betetzen duen hasieraketa: a:=0,

izan ere, $wp\left(a:=0,\ a^2 \leq n\right) = n \geq 0 = \phi$

Lortzen den programa, honako hau da:

<u>begin</u> $\{n \geq 0\}$

a:=0;

$\{a^2 \leq n\}$

<u>while</u> $(a+1)^2 \leq n$ <u>do</u> a:=a+1

<u>end</u> $\left\{a^2 \leq n < (a+1)^2\right\}$

Problema ebazteko modu hau erraza da, baina ez eraginkorra. Gatozen programa diseinatzera beste ideia batean oinarrituz: hurbilketa dikotomikoa.

Espezifikazioa, lehengo bera:

$\phi = n \geq 0$

$\psi = a^2 \leq n < (a+1)^2$,

baina oraingo inbariantea, beste hau:

$INB = a^2 \leq n \ \wedge \ b^2 > n$

eta

e=b-a.

Pentsa dezagun zein izan daitekeen hasieraketa egokia:

a:=0; b:=n. Zuzena al da?

$wp\left(a:=0;\ b:=n,\ a^2 \leq n \ \wedge \ b^2 > n\right) = n \geq 0 \ \wedge \ n^2 > n$

baina $n \geq 0$ izateak ez dakar derrigor $n^2 > n$ (n=0 edo n=1 bada $n^2 = n$ betetzen bait da).

Beraz, hasieraketa hori ez da zuzena, b-k hartzen duen balioa ez bait da behar bezain handia. Beste hasieraketa bat hartu beharko dugu.

1.- Hasieraketa:

a:=0; b:=n+1

$wp\left(a:=0;\ b:=n+1,\ a^2 \le n \wedge b^2 > n\right) = n \ge 0 \wedge (n+1)^2 > n$

Egiaz $n \ge 0$ aurreko baldintzak bermatzen du $(n+1)^2 > n$

2.- Iterazioaren B baldintza:

$\neg B = (b = a+1)$

$B = (b \ne a+1)$

$INB \wedge \neg B = \left(a^2 \le n \wedge b^2 > n \wedge b = a+1\right) \rightarrow \left(a^2 \le n < (a+1)^2\right)$

3.- Iterazioaren gorputza:

$\{a^2 \le n \wedge b^2 > n \wedge b \ne a+1\}$ *aldatu_a_eta_b* $\{a^2 \le n \wedge b^2 > n\}$

begin $\{a^2 \le n \wedge b^2 > n \wedge b \ne a+1\}$

d:= (a+b) div 2;

$\{a^2 \le n < d^2 \wedge d^2 \le n < b^2\}$

if $d^2 \le n$ then $\{d^2 \le n < b^2\}$

a:=d

else $\{a^2 \le n < d^2\}$

b:=d

end $\{a^2 \le n < b^2\}$

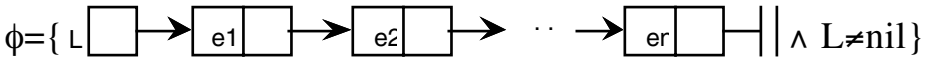
b-a adierazpena pausoero beheratzen da, a handitzen delako edo b beheratzen delako.

5.19 adibidea: *Erakuslez kateatutako L lista ez-hutsaren alderanzketa, erakusleen norantza aldatu besterik egin gabe.*

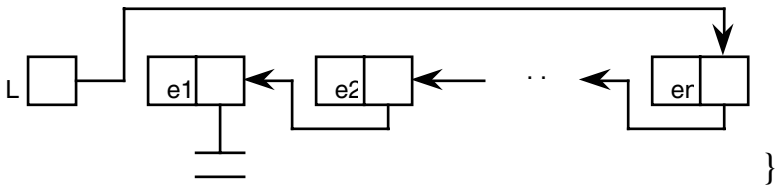
Jo dezagun lista honela erazagutu dela:

```
type korapilune = record
                      elem: T;
                      hurrengo: ↑korapilune
                  end;
var L: ↑korapilune
```

Aurre-ondoetako espezifikazioa:



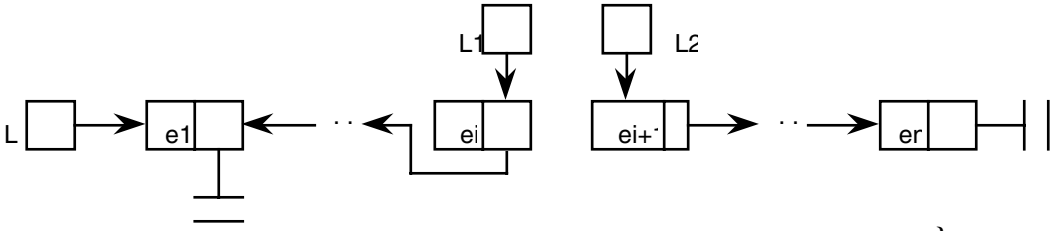
ψ={



}

Lista osoa alderanztu nahi badugu ere, alderanzketa pausoka-pausoka burutu beharko da eta bide horretako edozein tarteko egoeratan lista-zati bat alderanztua egongo da eta beste zati bat artean ukitu gabea. Ideia horrexek emango digu inbariantea finkatzeko oinarria. Dei diezaiegun L1 eta L2 bereizi ditugun lista-zatiei eta idatz dezagun inbariantea:

INB={ ∃i(1≤ i ≤ n ∧

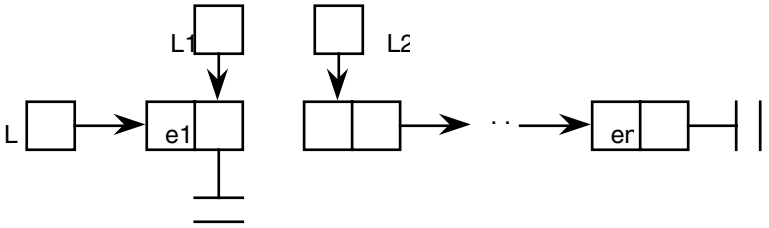


}

Borne-adierazpena, arrazonamendu horren haritik, alderanztu gabeko erakusleen kopurua izango da, hau da:

e=luz(L2)

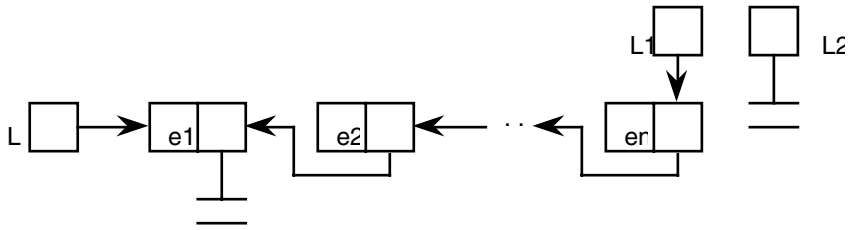
ϕ-tik hasi eta inbariantea beteko duen hasieraketa osatzeko, nahikoa da lista-zati alderanztua korapilune bakarrekoa dela suposatzea, hartara, hasieraketak bete beharko duen asertzioa inbariantearen kasu partikularra baizik ez da izango:



Hasieraketak agindu hauek bildu beharko ditu:

L1:=L;
L2:=L↑.hurrengo;
L↑.hurrengo:=nil;

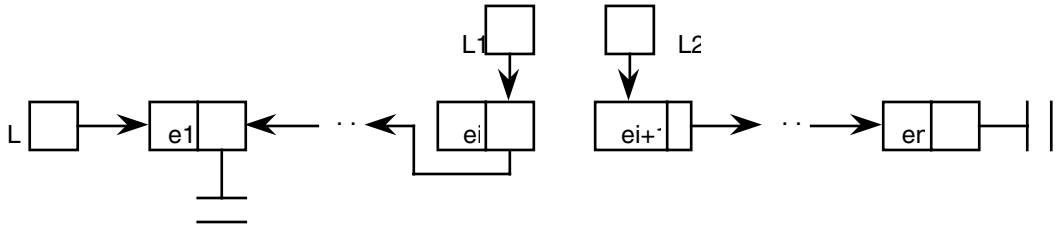
Iterazioaren bukaera-baldintza eratortzeko kontuan izan behar da ondoko baldintza bete dadin L2 listak hutsik geratu behar duela. Hau dela eta, B=(L2≠nil) hartzea da zentzuzkoena. Hala ere, besterik ezean, INB ∧ ¬ B baldintzaren ondorioz honako egoera hau suertatuko litzateke:



eta, agerikoa denez, ψ egiaz betetzeko L:=L1 aginduaren beharra dago bukaeran.

Azkenik, iterazioaren I gorputzaren eratorpenari ekin behar diogu. Dagoeneko ederki dakigunez, I-k inbariantea kontserbatu eta borne-adierazpenaren balioa txikitu egin behar du. Bi eginbehar hauek nabarmen beteko ditu ondoko espezifikazioari atxekitzen bazaio:

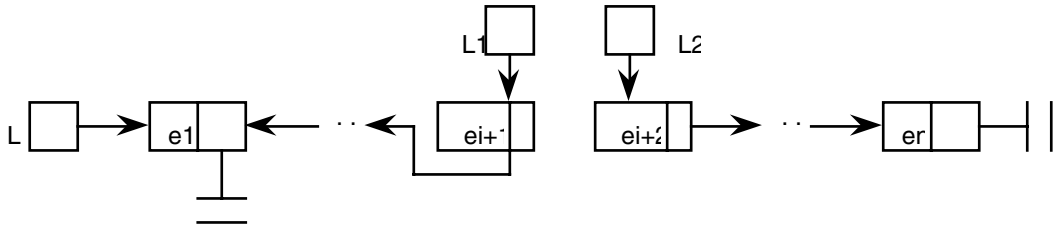
{



}

I

{



}

'Lagun' aldagai laguntzailea erabiliz espezifikazio hori beteko duen I agindu--multzoa izango da honakoa:

```
begin
  lagun:=L2↑.hurrengo;
  L2↑.hurrengo:=L1;
  L1:=L2;
  L2:=lagun
end;
```

Lortutako programa dokumentatua:

```
begin     {L=<e1,e2,...,en> ∧ L=nil}
 L1:=L; L2:=L↑.hurrengo; L↑.hurrengo:=nil;
 while L2≠nil do INB={L1=<ei,ei-1,...,e1>∧ L2=<ei+1,...,en>}
                  {e=luz(L2)}
          begin
                  lagun:=L2↑.hurrengo;
                  L2↑.hurrengo:=L1;
                  L1:=L2;
                  L2:=lagun
          end;
 L:=L1
end.      {L=<en,en-1,...,e1>}
```

Beste sententzia batzuk

Ikusi dugu orain arte nola erabil daitezkeen egiaztapeneko kontzeptuak programen diseinuan, baina aginduzko lengoaia baten oinarrizko eraikitzaileetara mugatu gara. Gainontzeko eraikitzaileak ere, orobat, antzeko ideiei jarraituz lor daitezke. Ikus ditzagun bakan batzuk:

Case agindua

Izan bedi agindu hau:

$\{\phi\}$ case e of $S_1:I_1;\ S_2:I_2;\ldots;S_k:I_k$ end $\{\psi\}$

Agindu honen semantika honako propietate hauez adieraz daiteke:

$$\phi \wedge e = S_1 \rightarrow wp(I_1, \psi)$$
$$\phi \wedge e = S_2 \rightarrow wp(I_2, \psi)$$
$$\ldots$$
$$\phi \wedge e = S_k \rightarrow wp(I_k, \psi)$$
$$\phi \rightarrow (e = S_1 \vee e = S_2 \vee \ldots \vee e = S_k)$$

Otherwise aukera duen case aginduarentzat, berriz:

$\{\phi\}$ case e of $S_1:I_1;\ S_2:I_2;\ldots;S_K:I_K$ otherwise $I_{K+1}\{\psi\}$

arauok bete behar dira:

$$\phi \wedge e = S_1 \rightarrow wp(I_1, \psi)$$
$$\phi \wedge e = S_2 \rightarrow wp(I_2, \psi)$$
$$\ldots$$
$$\phi \wedge e = S_k \rightarrow wp(I_k, \psi)$$
$$\phi \rightarrow e \neq S_1 \wedge \cdots \wedge e \neq S_k \rightarrow wp(I_{K+1}, \psi)$$

Repeat agindua

while aginduarekin egin dugun bezala inbariante bat eta borne-adierazpen bat aurkitu behar ditugu, eta hauetatik abiatuz iterazioa diseinatu. *{ϕ} repeat I until B {ψ}* betetzea nahi bada, INB inbariantea eta e borne-adierazpena emanda, repeat aginduaren gorputzak eta baldintzak ondoko bost propietateak errespetatu beharko dituzte:

1.– $\phi \rightarrow wp(I, INB)$

2.– $INB \wedge \neg B \rightarrow wp(I, INB)$

3.– $INB \wedge B \rightarrow \psi$

4.– $INB \rightarrow e > 0$

5.– $INB \wedge \neg B \wedge e = e_0 \rightarrow wp(I, e < e_0)$

For agindua

For aginduan exekuzio-tartea mugatzen denez inbariantea parametrizatu egin behar da, tarte hori korritzen duen aldagaiarekiko parametrizatu, hain zuzen ere. [ϕ,ψ] espezifikazioa beteko duen for agindua aurkitzeko eratorpen-estrategia ondokoa izango da:

1.- Izan bedi i ϕ-n edo ψ-n ageri ez den aldagaia, aurkitu P(i) propietatea.

2.- n_1 eta n_2 adierazpenak aurkitu, non:

(1) $\phi \wedge n_1 \leq n_2 \rightarrow P(aurre(n_1))$

(2) $P(n_2) \rightarrow \psi$

(3) $\phi \wedge n_1 > n_2 \rightarrow \psi$

3.- I errepikapenaren gorputza aurkitu, non:

$(P(aurre(i)) \wedge n_1 \leq i \leq n_2) \rightarrow wp(I, P(i))$

Metodoaren aplikazioak

5.20. adibidea: *z zenbaki oso positiboa emanda, sortu zenbaki hori bi karratuen batuketa gisa berridazteko era guztiak.*

Hasteko, problemaren aurre-ondoetako espezifikazioa azalduko dugu. Aurreko baldintza $\phi = z>0$ da. Ondoko baldintza idatzi aurrez, emaitza nolakoa izango den pentsatu behar da eta, horren arabera, zein objektu erabiliko ditugun erabaki. Jo dezagun emaitzak A eta B array-etan gordetzea erabakitzen dugula:

cons n = ?

var A,B: array [1..n] of integer;

non, $A[i]^2+B[i]^2=z$;

horietaz gain, k aldagai osoa ($1\le k\le n$) ere beharko dugu soluzio kopurua gordetzeko:

var k: integer;

$x^2+y^2 = y^2+x^2$ gertatzen denez, (x,y) eta (y,x) bikoteetatik bakarra gordetzea da zentzuzkoena. Nahasketarik sor ez dadin goranzko ordenan dauden bikoteak hobetsiko ditugu, hau da:

$$A[i]^2 + B[i]^2 = z \ \wedge \ 0 \le A[i] \le B[i]$$

Ezin litezke egon, horretara, lehenengo osagaia berdina duten bikoteak, ezta bigarren osagaia berdina dutenak ere:

$$\begin{array}{ccccccc} A[1] & < & A[2] & < \ \ldots \ < & A[k] \\ \le & & \le & & \le \\ B[1] & > & B[2] & > \ \ldots \ > & B[k] \end{array}$$

Ondoko baldintza labur-zehatza eta aldi berean argia izan dadin, ondoko laburdurok erabiliko ditugu:

soluzioak-dira(A, B, k) = $\forall i\left(1 \le i \le k \rightarrow \left(A[i]^2 + B[i]^2 = z \ \wedge \ 0 \le A[i \le B[i]]\right)\right)$

daudenak-dira(A, B, k) =

$$\forall h \forall p\left(h^2 + p^2 = z \ \wedge \ 0 \le h \le p \rightarrow \exists i\left(1 \le i \le k \wedge A[i] = h \wedge B[i] = p\right)\right)$$

Guztiz zehatzak izateko, errepikapenik ez dagoela eta, xeheago, A gorakorra dela gehitu beharko genuke:

gorakor(A, k) = $\forall i\left(1 \le i \le k \rightarrow A[i] < A[i+1]\right)$

Esanda bezala, B array-ak beste propietate hau betetzen du:

soluzioak-dira(A, B, k) $\wedge$ gorakor(A, k) $\rightarrow$ beherakor(B, k)

Honenbestez, aurre-ondoetako espezifikazioa idazteko prest gaude:

$\phi = \{z>0 \wedge n\ge 1\}$

$\psi = \{0\le k\le n \wedge$ soluzioak-dira(A, B, k) $\wedge$ daudenak-dira(A, B, k) $\wedge$ gorakor(A, k)$\}$

Gure hasierako ideia urratsero soluzio bat lortuko duen iterazioa eraikitzea da. Ebazpide horren arabera, inbariantea eratortzeko x aldagai berri bat erantsiz ahuldu beharko genuke ondoko baldintza:

INB= $0 \le k \le n \wedge$ soluzioak-dira(A, B, k) $\wedge$ artean-daudenak-dira(A, B, k, x) $\wedge$ gorakor(A, k)

non:

artean-daudenak-dira(A, B, k, x) =

$$\forall h \forall p \left(h^2 + p^2 = z \wedge 0 \le h \le p \wedge h < x \rightarrow \exists i \left(1 \le i \le k \wedge A[i] = h \wedge B[i] = p \right) \right)$$

Iterazioaren B baldintza boolearra asmatzeko ondoko inplikazioan oinarrituko gara:

INB $\wedge \neg$ B $\rightarrow$ daudenak-dira(A, B, k)

izan ere, A gorakorra bait da eta (A[i],B[i]) bikoteek $0 \le A[i] \le B[i]$ betetzen bait dute. Ondorioz, x aldagaiari $x^2+x^2 \le z$ eskatu behar zaio, hots:

B= $(2*x^2 \le z)$

$2*x^2 > z$ dela suposatuta, $x^2+y^2=z$ betetzen duen y-ren bat balego, $x^2+y^2<2*x^2$ gertatuko litzateke eta, hortaz, y<x. Hau da, soluzio hori jadanik A-n eta B-n gordeta edukiko genuke.

Horrelaxe bada:

INB $\wedge$ $2*x^2 > z \rightarrow$ daudenak-dira(A, B, k),

dudarik gabe, inbarianteak eta $2*x^2>z$ baldintzak ondoko baldintza beteko delako segurtasuna ematen digute.

Eratorri dugun programa-eskema:

```
begin   {z>0 ∧ n≥1}
        hasieraketa;
        while 2*x^2≤z do   {0≤k≤n ∧ soluzioak-dira(A, B, k) ∧
                           artean-daudenak-dira(A, B, k, x) ∧
                           gorakor(A, k)} e=z-x+1
        begin
                aurkitu-y (x,y,z);
                if badago-y (x,y,z) then
                                        begin
                                                k:=k+1;
                                                A[k]:=x;
                                                B[k]:=y
                                                end;
                x:=x+1
        end
end {0≤k≤n ∧ soluzioak-dira(A,B,k) ∧ daudenak-dira(A,B,k)∧
   gorakor(A, k)}
```

Algoritmo honetan idatzi ditugun ekintza abstraktuetatik bi finduko ditugu ondoren, eta bidenabar, asertzioekiko zuzenak direna egiaztatuko dugu:

hasieraketa: k:=0; x:=0;

honela bada, wp(k:=0; x:=0, INB) = n≥0, eta bistakoa denez hasierako baldintzatik formula hori ondoriozta daiteke: n≥1 → n≥0

aurkitu-y (x,y,z) ekintzak ondoko hirukotea bete beharko du:

{soluzioak-dira(A, B, k) ∧ artean-daudenak-dira(A, B, k, x) ∧ $2*x^2 \leq z$}

aurkitu-y (x,y,z)

{ ¬ badago-y(x,y,z) ∨
(soluzioak-dira((A,x/k+1), (B,y/k+1), k+1) ∧
artean-daudenak-dira((A,x/k+1), (B,y/k+1), k+1, x+1)) }

Kontua da 'y' aurkitu behar dugula $x^2+y^2=z \wedge 0 \leq x \leq y$ gerta dadin. Bestalde, 0≤k≤n baldintza arazorik gabe betetzeko n-ri balio egokia eman beharko diogu konstante-definizioan.

Ekintza abstraktuaren funtsezko eragina hauxe izango da:

{$2*x^2 \leq z$}

aurkitu-y (x,y,z)

$\{\neg$ badago-y(x,y,z) $\vee$ $(x^2+y^2=z \wedge 0 \leq x \leq y \wedge \forall h\left(x \leq h < y \rightarrow x^2 + h^2 \neq z\right))\}$

y≥x denez, x-tik hasi eta goranzko ordenan dagoen hurrengoaren bila saiatuko gara. Bilaketa hori burutuko duen iterazioaren inbariantea:

INB2 = INB ∧ $2*x^2 \leq z \wedge y \geq x \wedge \forall h\left(x \leq h < y \rightarrow x^2 + h^2 \neq z\right)$, non:

INB2 ∧ $x^2+y^2>z \rightarrow$ ¬ badago-y(x,y,z) gertatzen den batetik eta, INB2 ∧ $x^2+y^2=z$ betez gero, ondoko baldintzaren beste disjuntiba eragiten den bestetik.

Iterazioa bi egoeratan buka daiteke, $x^2+y^2=z$ edo $x^2+y^2>z$ denean. Garbi dago, beraz, iterazioaren B2 baldintza B2= $x^2+y^2<z$ izango dela, eta y gorantz aztertzen denez, borne-adierazpena: $\mathfrak{C}$ = z-y+1. Iterazioaren gorputza y:=y+1 izango da soilik.

Gainera INB ∧ $2*x^2 \leq z$ baldintzak ez du esan nahi INB2 beteko denik nahitaez, ez bait dago esaterik $y \geq x \wedge \forall h\left(x \leq h < y \rightarrow x^2 + h^2 \neq z\right)$ gertatuko denik. Beraz, iterazioaren aurretik hasieraketa bat jarri behar dugu baldintza hauek zierto bete daitezen. y:=x asignazioa hasieraketa egokia da, izan ere:

wp(y:=x, $y \geq x \wedge \forall h\left(x \leq h < y \rightarrow x^2 + h^2 \neq z\right)$)=true.

aurkitu-y (x,y,z) ekintza abstraktua honela findu daiteke:

begin $\{2*x^2 \le z \ \wedge \ INB\}$
y:=x;
while x2+y2<z do
$\{INB \wedge 2*x^2 \le z \wedge y \ge x \wedge \forall h(x \le h < y \rightarrow x^2 + h^2 \ne z)\}$
$\{e2 = z - y + 1\}$
y:=y+1
end $\{x^2 + y^2 > z \ \vee \ (x^2 + y^2 = z \wedge 0 \le x \le y \wedge \forall h(x \le h < y \rightarrow x^2 + h^2 \ne z))\}$

Eratorpenaren azken emaitza programa hau da:

begin $\{z > 0 \ \wedge \ n \ge 1\}$
k:=0; x:=0;
while $2*x^2 \le z$ do INB={0≤k≤n ∧ soluzioak-dira(A, B, k)
∧ artean-daudenak-dira (A, B, k, x) ∧
gorakor(A, k)} {e=z-x+1}
begin $\{2*x^2 \le z \ \wedge \ INB\}$
y:=x;
while x2+y2<z do
INB2=$\{INB \wedge 2*x^2 \le z \wedge y \ge x \wedge \forall h(x \le h < y \rightarrow x^2 + h^2 \ne z)\}$
$\{e2 = z - y + 1\}$
y:=y+1;
$\{x^2 + y^2 > z \ \vee \ (x^2 + y^2 = z \wedge 0 \le x \le y \wedge \forall h(x \le h < y \rightarrow x^2 + h^2 \ne z))\}$
if $x^2+y^2=z$ then
begin
k:=k+1;
A[k]:=x;
B[k]:=y
end;
{soluzioak-dira(A, B, k) ∧
artean-daudenak-dira(A,B,k, x+1) ∧ gorakor(A,k)}
x:=x+1
end
end {0≤k≤n ∧ soluzioak-dira(A,B,k) ∧ daudenak-dira
(A, B, k) ∧ gorakor(A, k)}

5.21. adibidea: *Hamming-en sekuentzia: Sortu goranzko ordenan HS-ren (Hamming-en sekuentziaren) lehenengo n zenbakiak. HS sekuentzia soilik 2, 3 edo 5 zenbaki lehenez zatigarriak diren zenbakiek osatzen dute.*

HS = 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, ...

Honela defini daiteke sekuentzia hau:

a) $1 \in HS$

b) Baldin $x \in HS$ orduan $2*x$, $3*x$, $5*x \in HS$

c) HS-n ez dago beste zenbakirik

Eman dezagun, hasteko, sekuentzia A[1..n] osozko array-an gordetzen dela, eta idatz dezagun horren araberako aurre-ondoetako espezifikazioa:

ϕ= {n≥1}

ψ= {A[1..n]-k HS-ren lehenengo zenbakiak dauzka, non HS goian definitu dugun sekuentzia den}

Espezifikazioak horretara behartzen ez bagaitu ere, arrazoizkoena goranzko ordenan sortzea dela dirudi. Hasierako ideia iterazio-pausoero A array-ari elementu bat eranstea da, hori bai, goranzko ordenari eutsiz. Ideia hori gauzatzeko inbariante egokia izango da honakoa:

INB= {A[1..k]=HS-ren lehenengo k zenbakiak ∧ 1≤k≤n}

Iterazioa k=n gertatutakoan bukatuko da, hau da, iterazioaren baldintzatzat k≠n hartuko dugu eta $\mathfrak{C}$=n-k borne-adierazpentzat. Estraineko aldian inbariantea bete dadin k:=1 aginduaz programa hasieratzea premiazkoa da. Baina, nahikoa al da hasieraketa hori?

Ikus dezagun:

wp(k:=1, A[1..k]=HS-ren lehenengo k zenbakiak ∧ 1≤k≤n) = (A[1]=1 ∧ n≥1)

Bistan da hasieraketan bi aginduok bildu behar direla:

A[1]:=1; k:=1;

Orainokoan ez dugu aparteko zailtasunik topatu. Iterazioaren gorputza eratortzea, ordea, ez da hain erraza. Pausoero x txikiena aurkitu beharko dugu non x>A[k] eta $x \in HS$, eta aurkitutakoan A[k+1]-ean gorde.

Badakigu aurkitu nahi dugun x zenbakia $2*y$, $3*y$ edo $5*y$ erakoa dela $y \in A[1..k]$ batentzat, eta gainera x>y. Ez ote da egokia ondoko ideia hau?

" A[1..k] emanda, izan bedi

B[1..n]= {z / $y \in$A[1..k], z>A[k], non $z=2*y$ edo $z=3*y$ edo $z=5*y$},

orduan: x= min{z / $z \in$B[1..n]}".

Ikusi batean ez dirudi eraginkorra izango denik, n nahi bezain handia izan bait daiteke eta minimoa aukeratzeko B-k ordenatuta egon behar bait du.

Ea bada, azter dezagun beste ideia hau:

"A[1..k] emanda, izan bitez:

x2= { z / z=2∗y non y∈A[1..k] }
x3= { z / z=3∗y non y∈A[1..k] }
x5= { z / z=5∗y non y∈A[1..k] };

pausoero erantsi beharreko elementua hiru horietatik minimoa da". Kontuan hartu behar da hirurak A[k] baino hertsiki handiagoak direla.

A[1..k] array-a eta x2, x3 eta x5 balioak lotzen dituen propietatea P(A[1..k],x2,x3,x5) notazioaz laburtuko dugu.

Inbariantea:

INB={A[1..k]=HS-ren lehenengo k zenbakiak ∧ P(A[1..k],x2,x3,x5) ∧ 1≤k≤n}

Inbariantean egin dugun eransketa berri honek hasieraketa, iterazio-baldintza eta borne-adierazpena birplanteatzera behartzen gaitu: k≠n baldintza eta n-k adierazpena dauden horretan utz daitezkeen arren, inbariantean ageri diren aldagai berriak hasieratu egin beharko ditugu. Horretarako, inbariantearen eta geneukan hasieraketaren aurreko baldintza ahulena lortuko dugu:

wp(A[1]; k:=1, HAS) = (A[1]=1 ∧ x2=min{2} ∧ x3=min{3} ∧ x5=min{5} ∧ n≥1)

Beraz, hasieraketa:

A[1]:=1; k:=1; x2:=2; x3:=3; x5:=5;

Honatx, orain arte lortu dugun programa-eskema:

```
begin     {n≥1}
  A[1]:=1; k:=1; x2:=2; x3:=3; x5:=5;
        INB={A[1..k]=HS-ren lehenengo k zenbakiak ∧
              P(A[1..k],x2,x3,x5) ∧ 1≤k≤n}
  while k≠n do
        begin
              k:=k+1;
              A[k]:=min(x2,x3,x5);
              birkalkulatu(x2,x3,x5)
        end
end           {A[1..n]=SH-ren lehenengo n zenbakiak}
```

Eginkizun daukagu oraindik bi aginduren finketa, iterazioaren gorputzak inbariantea kontserba dezan:

```
begin   {A[1..k]=HS-ren lehenengo k zenbakiak ∧
              P(A[1..k],x2,x3,x5) ∧ 1≤k<n}
k:=k+1;
{A[1..k-1]=HS-ren lehenengo k-1 zenbakiak ∧
P(A[1..k-1],x2,x3,x5) ∧ 1≤k≤n}
 A[k]:=min(x2,x3,x5);
{A[1..k]=HS-ren lehenengo k zenbakiak ∧
P(A[1..k-1],x2,x3,x5) ∧ 1≤k≤n}
 birkalkulatu(x2,x3,x5)
end     {A[1..k]=HS-ren lehenengo k zenbakiak ∧
        P(A[1..k],x2,x3,x5) ∧ 1≤k≤n}
```

Gatozen lehengoarekin:

```
{A[1..k-1]=HS-ren lehenengo k-1 zenbakiak ∧
P(A[1..k-1],x2,x3,x5) ∧ 1≤k≤n}
        A[k]:=min(x2,x3,x5);
{A[1..k]=HS-ren lehenengo k zenbakiak ∧
P(A[1..k-1],x2,x3,x5) ∧ 1≤k≤n}
```

Hiru zenbakiren minimoa baldintzazko egitura batez hauta daiteke. Baina, horrez gain, ez da ahantzi behar x2, x3 eta x5 multzo banaren minimoak direna, eta hiruretatik edozein hartuta ere, ondokoa betetzen dela: baldin xi=min{x2,x3,x5}, orduan, x2, x3 eta x5 balioen definizioa dela eta, xi=min{z / z∈HS ∧ z>A[k]}.

Hau da:

xi=min{x2,x3,x5} ∧ P(A[1..k-1],x2,x3,x5) → xi=min{z / z∈HS ∧ z>A[k]}

Esandakotik honakoa erator dezakegu:

```
 {A[1..k-1]=HS-ren lehenengo k-1 zenbakiak ∧
 P(A[1..k-1],x2,x3,x5)}
if (x2≤x3 and x2≤x5) then       {x2=min{z / z∈HS ∧ z>A[k]}
                                                A[k]:=x2
else if (x3≤x2 and x3≤x5) then  {x3=min{z / z∈HS ∧ z>A[k]}
                                                A[k]:=x3
   else if (x5≤x2 and x5≤x3) then{x5=min{z / z∈HS ∧ z>A[k]}
                                                A[k]:=x5;
{A[1..k]=HS-ren lehenengo k zenbakiak}
```

Bukatzeko, *birkalkulatu* abstrakzio funtzionala findu beharrean gaude:

`{A[1..k]=HS-ren lehenengo k zenbakiak ∧`
`P(A[1..k-1],x2,x3,x5)}`
birkalkulatu(x2,x3,x5)
`{P(A[1..k],x2,x3,x5)}`

Aldez aurretik P(A[1..k],x2,x3,x5) betetzeak ez du hiruretatik soilik bakarra aldatu behar denik esan nahi. Eguneraketa burutzerakoan beste baldintza hauei ere erreparatu behar zaie zein aldatu jakiteko:

A[k]≥x2 ⇒ x2 aldatu
A[k]≥x3 ⇒ x3 aldatu
A[k]≥x5 ⇒ x5 aldatu

Gainera, A[1..k] gorakorra denez eta HS-ren lehenengo k zenbakiak dauzkanez, x2=2∗A[i] betetzen da i-ren batentzat, 1≤i≤k. Horretara, x2-ren balio berria 2∗A[i+1] izango da, eta beste horrenbeste gertatuko da x3 eta x5 balioentzat. Hots:

gorakor(A[1..k]) ∧ x2=2∗A[i_2] → 2∗A[i_2+1]= min{ z / z=2∗y eta y∈A[1..k] eta z>x2}
gorakor(A[1..k]) ∧ x3=3∗A[i_3] → 3∗A[i_3+1]= min{ z / z=3∗y eta y∈A[1..k] eta z>x3}
gorakor(A[1..k]) ∧ x5=5∗A[i_5] → 5∗A[i_5+1]= min{ z / z=5∗y eta y∈A[1..k] eta z>x5}

Horren arabera bi aukera dauzkagu: i_2, i_3 eta i_5 indizeak pausoero gorde, edo x2, x3 eta x5 birkalkulatzean bilatu. Lehenengo aukera hobetsiko dugu, horrela eratorritako programa eraginkorragoa izango da eta. Hala ere, aukeraketa horrek diseinuari berriz ere errepaso bat ematea eskatzen du:

INB={A[1..k]=HS-ren lehenengo k zenbakiak ∧ P(A[1..k],x2,x3,x5) ∧ 1≤k≤n ∧
x2=2∗A[i_2] ∧ x3=3∗A[i_3] ∧ x5=5∗A[i_5] }

e=n-k (lehen bezala)

Inbariantean hiru aldagai berri ageri direnez, hasieraketa egokitzera behartuta gaude:

wp(A[1]:=1; k:=1, INB) =
(A[1]=1 ∧ x2=2 ∧ x3=3 ∧ x5=5 ∧ n≥1 ∧ A[i_2]=1 ∧ A[i_3]=1 ∧ A[i_5]=1)

Aurreko baldintza ahulenak erakusten digu i_2,i_3 eta i_5 aldagaiei 1 balioa esleitu behar zaiela hasieran, bestela ez bait litzateke inbariantea beteko iterazio-sarreran.

Hona bada hasieraketa:

begin A[1]:=1; x2:=2; x3:=3; x5:=5; i_2:=1; i_3:=1; i_5:=1 end

Hiru aldagai berriok nahiz inbariantean erantsitako propietateak ez dute zertan *A[k]:=min(x2,x3,x5)* abstrakzioa aldatu behar. Orain dugu egokiera *birkalkulatu(x2,x3,x5)* abstrakzioa fintzeko, baina kontuan izan behar da i_2, i_3 eta i_5 indizeak ere eguneratu beharko dituela, inbariantean ezarri dugun baldintza berria bete dadin:

{A[1..k]=HS-ren lehenengo k zenbakiak ∧ P(A[1..k-1],x2,x3,x5) ∧ 1≤k≤n ∧
x2=2∗A[i2] ∧ x3=3∗A[i3] ∧ x5=5∗A[i5] }

birkalkulatu(x2,x3,x5)

{A[1..k]=HS-ren lehenengo k zenbakiak ∧ P(A[1..k],x2,x3,x5) ∧ 1≤k≤n ∧
x2=2∗A[i2] ∧ x3=3∗A[i3] ∧ x5=5∗A[i5] }

Punturik azpimarragarrienak nabarmenduz:

{P(A[1..**k-1**],x2,x3,x5) ∧ x2=2∗A[i2] ∧ x3=3∗A[i3] ∧ x5=5∗A[i5] }

birkalkulatu(x2,x3,x5)

{P(A[1..**k**],x2,x3,x5) ∧ x2=2∗A[i2] ∧ x3=3∗A[i3] ∧ x5=5∗A[i5] },

eta ekintza abstraktu horren finketak ez du zailtasun berezirik, aipatu ditugun propietateak gogoan edukiz eta 'xi' horiek soilik A[k]=xi gertatzen denean aldatu behar direla ahantzi gabe, nahiz eta badakigun A[k]=xi berdintza behin baino gehiagotan gerta daitekeela aldi berean.

5.22. adibidea: *Hurrengo permutazioaren problema.*

Izan bedi n digituko zenbakia (n≥2) A[1..n] array-an adierazia. A[1] elementua mailarik handieneko digitua izango da. Adbidez, n=6 hartuta, A=(1,2,3,5,4,2) array-ak 123.542 zenbaki osoa adierazten du.

Aren "hurrengo permutazioa" digitu berberez osatutako hurrengo zenbaki handiagoa da, aurreko adibidean: (124.235). Hurrengorik egon dadin ezinbestekoa da A ez egotea beheranzko ordenan.

A eta n emanda, non A ez den beherakorra, diseinatu An hurrengo permutazioa lortzen duen programa.

Espezifikazioa

$$\phi = \left(n \geq 2 \wedge A = (a_1,\ldots,a_n) \wedge \neg \forall k(1 \leq k < n \rightarrow A[k] \geq A[k+1])\right) =$$
$$\left(n \geq 2 \wedge A = (a_1,\ldots,a_n) \wedge \exists k(1 \leq k < n \wedge A[k] < A[k+1])\right)$$

$$\psi = \left(A = hurr\_perm(a_1,\ldots,a_n)\right), \text{ non:}$$
$$hurr\_perm(a_1,\ldots,a_n) = (b_1,\ldots,b_n) \wedge (b_1,\ldots,b_n) = perm(a_1,\ldots,a_n) \wedge$$
$$\forall B\left(B = perm(a_1,\ldots,a_n) \wedge \sum_{i=1}^{n} B[i]*10^{(n-i)} > \sum_{i=1}^{n} a_i*10^{(n-i)} \rightarrow \sum_{i=1}^{n} B[i]*10^{(n-i)} \geq \sum_{i=1}^{n} b_i*10^{(n-i)} \right)$$

Zehaztapen honek iradokitzen digun lehenengo ideia permutazio guztiak sortzea da, hortara denen artean hurrengoa aukeratzeko. Hala ere, ebazpide eraginkorragoa pentsatu behar dugu, ez bait dirudi oso egokia denik permutazio guzti-guztiak sortu behar izatea.

Baina, nola jakin zein den hurrengo permutazioa?

A daukagula, nola asmatu zein den hurrengoa?

Ondoko lau propietateok argituko digute hurrengo permutazioa lortzeko bidea:

1) $\neg \text{beherakor}(A) \rightarrow \exists i\left(1 \le i < n \wedge A[i] < A[i+1]\right)$
2) A zenbakia hazteko handitu daitekeen maila txikieneko digitua, 1) gertatzen deneko A[i] maila txikienekoa izango da.
3) A[i] elementua bera baino handiagoa den A[i+1..n] horien arteko digiturik txikienarekin permutatu behar da.
4) A[i+1..n] sekzioa beherakorra da 1) eta 2)-tik ondoriozta daitekeenez. 3)-n deskribatu den permutazioa burutu ondoren A[i+1..n] sekzioak beherakor izaten segituko du. Beraz, ahalik eta txikiena izan dadin beharrezkoa da sekzio hori alderanztea.

Esandakoak ez dira hain begibistakoak. Bereziki, 3. propietateko permutaketak A[i+1..n] sekzioaren beherakortasuna mantentzen duela ondo pentsatu beharreko baieztapena da.

Ondoko baldintza honek hobeto isladatzen du lau propietateen araberako hurbilpena:

$$\psi = \left(A = \text{hurr\_perm}\left(a_1,\ldots,a_n\right)\right), \text{ non:}$$

$$\text{hurr\_perm}\left(a_1,\ldots,a_n\right) = \left(b_1,\ldots,b_n\right) \wedge \left(b_1,\ldots,b_n\right) = \text{perm}\left(a_1,\ldots,a_n\right) \wedge$$
$$\left(a_i < a_{i+1} \wedge \text{beherakor}\left(\left(a_{i+1},\ldots,a_n\right)\right) \wedge a_k = \min\left\{x \in \left(a_{i+1},\ldots,a_n\right) / x > a_i\right\}\right.$$
$$\left.\wedge\left(b_1,\ldots,b_n\right) = \left(a_1,\ldots,a_{i-1},a_k,a_n,\ldots,a_{k+1},a_i,a_{k-1},\ldots a_{i+1}\right)\right.$$

Honenbestez, programaren eskema honelakoa litzateke:

$$
\begin{array}{ll}
\underline{\text{begin}} & \left\{n \geq 2 \wedge A = (a_1,\ldots,a_n) \wedge \exists k(1 \leq k < n \wedge A[k] < A[k+1])\right\} \\
& \quad i\_\text{aurkitu}\ (A,i); \\
& \qquad \left\{a_i < a_{i+1} \wedge \text{beherakor}((a_{i+1},\ldots,a_n)) \wedge 1 \leq i < n\right\} \\
& \quad k\_\text{aurkitu}\ (A,i+1,k); \\
& \qquad \left\{\text{beherakor}((a_{i+1},\ldots,a_n)) \wedge a_k = \min\left\{x \in (a_{i+1},\ldots,a_n) \,/\, x > a_i\right\}\right\} \\
& \quad \text{permutatu}\ (A[i],A[k]); \\
& \qquad \left\{A[1..n] = (a_1,\ldots,a_{i-1},a_k,a_{i+1},\ldots,a_{k-1},a_i,a_{k+1},\ldots a_n)\right\} \\
& \quad A\_\text{alderantuz}\ (A,i+1,n) \\
\underline{\text{end}} & \quad \left\{A[1..n] = (a_1,\ldots,a_{i-1},a_k,a_n,\ldots,a_{k+1},a_i,a_{k-1},\ldots a_{i+1})\right\}
\end{array}
$$

Algoritmo honetan erabili ditugun ekintza abstraktuak findu beharko genituzke orain. Asertzioek ematen diguten informaziotik ez da zaila formalki eratortzea, eta ariketa interesgarria da gainera.

Ariketak

Eratorri formalki ondoko ekintza abstraktuak:

5.3. A array-aren elementuak binaka permutatu.

5.4. Input-ean beheranzko ordenan dagoen lehen zenbaki osoen bikotea bilatu.

5.5. A eta B array-ak emanda, Bn ere badagoen Ako lehen elementua aurkitu.

5.6. Ondoko espezifikazioa betetzen duen P programa:

{n≥1} P {PAL=T ↔ ∀i (1≤i≤n div 2 → A[i] = A[n-i+1])}

5.7. A eta B array-etan (A,B: array [1..n] of 0..1) zenbaki bitar bana errepresentatzen da, eta iterazio baten bidez A eta Bren bi oinarriko batura lortu nahi da S array-an (S: array [0..n] of 0..1), D array laguntzailea (D: array [0..n] of 0..1) erabiliz digituen baturek sortzen dituzten burukoak gordetzeko. Hauxe da espezifikazioa:

{n≥1}

P

{ ∀i (1≤i≤n → S[i] = (A[i]+B[i]+D[i]) mod 2)
∧ S[0] = D[0]
∧ ∀j (0≤j<n → D[j] = (A[j+1]+B[j+1]+D[j+1]) div 2)
∧ D[n]=0 }

5.8. A[1..n] array-an elkarren ondoko bi zenbakien artean dagoen diferentziarik handiena (balio absolutua) idatzi. Hau da:

{n≥2}

P

{output=<| A[i] -A[i+1] |> ∧ 1≤i<n ∧ ∀k (1≤k<n → | A[k] -A[k+1] | ≤ | A[i] - A[i+1] |) }

5.9. A[1..n] array-a eta **z** balio osoa emanda, ph aldagaian itzuli ondoko espezifikazioa betetzen duen emaitza.

{n≥1}

P

{∀i (1≤i≤n → |A[i] - z| ≥ |A[ph] - z|) }

5.10. x emanda eta A[0..n] array-an polinomio baten koefizienteak dauzkagula, x jakin horrentzat polinomioaren balioa kalkulatu, hau da:

{n≥0}

P

{z=A[n]∗x^n+A[n-1]∗x^{n-1}+...+A[1]∗x+A[0] }

5.11. Zenbaki oso bat eta B oinarria emanda , idatzi zenbaki hori B oinarrian.

6. ALGORITMO ERREKURTSIBOAK

6.1. INDUKZIO ESTRUKTURALA

Zenbaki arrunten propietateak frogatzeko metodo aski ezaguna da indukzioaren printzipioan oinarritutakoa. Printzipio hori era askotan formulatzen da, eta honako hau sinpleena da:

$$\Big(P(0) \wedge \forall n\big(P(n) \rightarrow P(n+1)\big)\Big) \rightarrow \forall n\big(P(n)\big)\Big)$$

edo, euskaraz esanda, zenbaki arruntek P propietatea betetzen dutela frogatzeko nahikoa dela:

a) P(0), hau da, "0 zenbakiak P betetzen du". Frogapenaren atal honi *oinarrizko pauso* deritzo.
b) P(n) betetzen dela jota (indukzio-hipotesia), P(n+1) ere betetzen da. *Indukzio pauso* izendatu ohi da bigarren hau.

Batzuetan, ez dugu frogatu nahi izango P propietatea zenbaki arrunt guztiek betetzen dutela, baizik eta bakarrik n_0 jakin bat baino handiagoek: $\forall n\big(n \geq n_0 \rightarrow P(n)\big)$. Kasu honetan burutu beharreko pausoak:

a) P(0), eta
b) $\forall n\big(n \geq n_0 \wedge P(n) \rightarrow P(n+1)\big)$

Beraz, metodoa indukzioaren printzipioaren ondoko formulazioan oinarritzen da:

$$\Big(P(n_0) \wedge \forall n\big(n \geq n_0 \wedge P(n) \rightarrow P(n+1)\big)\Big) \rightarrow \forall n\big(n \geq n_0 \rightarrow P(n)\big)\Big)$$

Formulazio hori aurrekoaren orokorpena besterik ez da. Izan ere, hasiera-hasieran formulatu dugun indukzioaren printzipioa n_0=0 deneko kasua bait da.

Oraindik, hala ere, zenbait propietate frogatzeko motz geratzen dira ikusi ditugun formulazioak. Orain artekoetan indukzio-hipotesi gisa aurreko zenbaki arruntak P propietatea betetzen duela esaten da, eta ez besterik. Baina zenbaitetan juxtu aurrekoak ezezik propietatea betetzen dutenak aurreko zenbaki batzuk direla suposatu behar izaten da. Aberasketa horrek printzipioa birformulatzeko parada eskainiko digu.

Indukzioaren printzipio osotua zenbaki arruntak $\leq$ ordena partzialaren arabera ordenatuta egotean oinarritzen da, eta hauxe da formulaziorik erabilkorrena:
Zenbaki arruntek P propietatea betetzen dutela egiaztatzeko nahikoa da ondokoa frogatzea:

a) Oinarrizko pausoa: P(0), eta
b) Indukzio-pausoa: $\forall k\left(k \leq n \rightarrow P(k)\right)$ betetzen dela jota, P(n+1) ere betetzen da.

Bi atalak bilduz:

$$\left(P(0) \wedge \forall n\left(\forall k\left(k \leq n \rightarrow P(k)\right) \rightarrow P(n+1)\right) \rightarrow \forall n\, P(n)\right)$$

Lehen egin dugunaren antzera, azken formulazio hau ere orokortu egin daiteke, n_0-tik aurrerako zenbaki arruntek P propietatea betetzen dutela frogatu nahi izanez gero, hots, $\forall n\left(n \geq n_0 \rightarrow P(n)\right)$:

$$\left(\forall n\left(\forall k\left(n_0 \leq k < n \rightarrow P(k)\right) \rightarrow P(n)\right) \rightarrow \forall n\left(n \geq n_0 \rightarrow P(n)\right)\right)$$

Indukzioaren printzipioak, gainera, $\dot{.}$-ren gaineko eragiketak definitzeko ere balio du. Printzipio honetan oinarritutako definizioak induktiboak direla esaten da. Adibide batzuk:

(1) Faktorialaren definizioa:

fakt: $\dot{.} \rightarrow \dot{.}$
$n \rightarrow n! = n*(n-1)*\ldots 3*2*1$

fakt(0) = 1
fakt(n) = n∗fakt(n-1) baldin n≥1
edo
fakt(0) = 1
fakt(n+1) = (n+1)∗fakt(n)

(2) Berreketaren definizioa:

berre: $\dot{.}\,x\,\dot{.} \rightarrow \dot{.}$
$(n,m) \rightarrow n^m$

berre(n,0) = 1
berre(n,m) = n∗berre(n,m-1) baldin m≥1
edo
berre(n,0) = 1
berre(n,m+1) = n∗berre(n,m)

non indukzioa funtzioaren bigarren argumentuari aplikatzen zaion.

Definizio induktiboaren izena, Matematiketan maiztto erabilia, definizioen formari zor zaio. Izan ere, indukziozko frogapenetan bezala, definizio hauetan gutxienez oinarrizko pauso bat bai bait dago, funtzioa esplizituki definitzen denekoa (fakt(0)=1, berre(n,0)=1) ,

eta baita indukzio-pausoren bat ere, non funtzioa bere buruaz baliatuta definitzen bait da, betiere argumentu txikiagoetarako (fakt(n) = n∗fakt(n-1), berre(n,m) = n∗berre(n,m-1)).

Informatikaren arloan definizio-eredu honi errekurtsibo esan ohi zaio. Hain zuzen ere, horretan oinarritzen da diseinu errekurtsiboa.

Aurrera joz, esan dezagun indukzio estrukturala $.^{\cdot}$-ren gaineko indukzioaren orokorketaz lortzen dela. Azken batean, indukzioaren printzipioaren oinarrian $.^{\cdot}$ multzoa 0-tik hasi eta *ondorengo* eragiketaz sortzen den egitura delako egitatea dago.

$.^{\cdot}$ multzoa, hortaz, 0 eta *ondorengo* eragiketaz eraikitzen da:

$$0^{\bullet} \rightarrow \text{ond}(0)^{\bullet} \rightarrow \text{ond}^2(0)^{\bullet\bullet\bullet} \rightarrow^{\bullet\bullet\bullet} \text{ond}^n(0)\ ^{\bullet} \rightarrow \text{ond}^{n+1}(0)^{\bullet\bullet\bullet}$$

$.^{\cdot}$ egitura, honako propietate hauek betetzen dituen S multzorik txikiena da:

a.- $0 \in S$

b.- Edozein $n \in$ S-rentzat ondorengo(n)$\in$S

Hori dela eta, $.^{\cdot}$-k propietate bat betetzen duela frogatzea, 0 zenbaki arruntak bete eta ondorengo eragiketak kontserbatu egiten duela frogatzea besterik ez da, Era berean, $.^{\cdot}$-ren gaineko eragiketak indukzioz definitzeko 0-rentzat definitzen dira lehenbizi eta, n-ren gaineko balioaren arabera, ondorengo(n)-rentzat gero. Eredu horretako definizioak ditugu honako hauek:

x+0 = x
x+ondoren(y) = ondoren(x+y)

x∗0 = 0
x∗ondoren(y) = x+x∗y

$x^0 = 1$
$x^{\text{ondoren(y)}} = x * x^y$

Horrela bada, multzo infinitu baten definizioan oinarrizko objektuen multzoa eta objektu sinpleagoetatik multzoko beste objektuak sortzen dituzten eragiketa eraikitzaileen multzoa adierazten badira, orduan multzo horren gaineko eragiketak indukzio bidez defini daitezke, eta definitu ezezik, multzoko objektuek eta definitutako funtzioek betetzen dituzten propietateak frogatu ere bai.

Formalago esanda:

A multzoaren definizio induktiboa: Izan bedi B *oinarrizko multzo* deitutako multzo ez-hutsa eta izan bedi K *eraikitzaileen multzoa*, hau da, A eremukoa duten eta heina ere A duten funtzioen multzoa. A multzoa S multzorik txikiena izango da, non:

1.- $B \subseteq S$

2.- Edozein $r \in$ K-rentzat (n argumentukoa) eta edozein $a_1,...,a_n \in$ S-rentzat:
$r(..., a_1,..., a_n,...) \in S$

6.1. *adibidea*: *WP={P / P while-programa da} multzo infinitua indukzio estrukturalaren bidez defini daiteke:*

B={**x:=t** / x∈ Ald , t∈Gai}
K={ _;_ , **if-then-else** , **while-do**}

edo, beste era batera:

1.- **x:=t** ∈ WP baldin x∈ Ald ∧ t∈Gai
2.- P1**;**P2 ∈ WP baldin P1,P2∈ WP
3.- **if** b **then** P1 **else** P2 ∈ WP baldin P1,P2∈ WP ∧ b∈Bool-adiera
4.- **while** b **do** P ∈ WP baldin P∈ WP ∧ b∈Bool-adiera

6.2. *adibidea*: *T^*={<s_1,...,s_n> / $s_i \in T$ ∧ i=1,...,n ∧ n≥0} honela defini daiteke:*

B={**< >**}
K={**erantsi**: T^*x$T \rightarrow T^*$ (elementu baten eransketa sekuentzia batean)}

6.3. *adibidea*: *T^+={<s_1,...,s_n> / $s_i \in T$ ∧ i=1,...,n ∧ n≥1} honela defini daiteke:*

B={**<x>** / x ∈T}
K={**erantsi**: T^+x$T \rightarrow T^+$ (elementu baten eransketa sekuentzia batean)}

Multzo baten definizio induktiboak multzo hori eremutzat hartuko duten funtzioak definitzeko bidea ematen du.

Funtzioen definizio induktiboa: Izan bedi A indukzioz definitutako multzoa, non B oinarrizko multzoa den eta K eraikitzaileen multzoa; A-n (A behinik-behin egongo da eremuan, nahiz eta beste multzoren bat ere egon daitekeen) f funtzioa indukzioz definitzeko beharrezkoa da ondokoak finkatzea:

1.- Edozein b ∈ B-rentzat: f(..., b,...)
2.- Edozein r ∈ K-rentzat eta edozein a_1,...,a_n ∈ A-rentzat:
  f(...,r(...,a_1,...,a_n,...),...);
  f(...,a_1,...), f(...,a_2,...) eta f(...,a_n,...) funtzioen arabera.

Era honetako definizioak funtsik izango badu A-ko edozein a baliok irudi bakarra izan behar du. Hala gertatzen bada f hori ondo definituta egongo da. Dena den, ustekabekorik gerta ez dadin, funtzio bat indukzioz ondo definituta dagoen ala ez jakiteko bada irizpide zehatzagorik:

Multzo baten definizio induktibo librea: A multzoaren definizio induktiboa, B oinarrizko multzoaren eta K eraikitzaileen multzoaren bidezkoa, librea izango da edozein a∈A-rentzat a∈B bada edota, bestela, existitzen bada r funtzio eraikitzaile bakarra (r∈K) eta a_1,...,a_n n-kote bakarra (a_1,...,a_n∈A) non a=r(a_1,...,a_n).

Definizio honek aukera ematen digu ondo definitutako funtzioei buruz ondokoa baieztatzeko:

A multzoaren definizio induktiboa librea bada, A-n indukzioz definitutako funtzioa (goian ikusitako definizio-eredua errespetatuz) ondo definituta egongo da.

6.4. *adibidea*: *Defini dezagun while-programen gainean ondoko funtzioa:*

wkop: $WP \rightarrow .^{\cdot}$

$p \rightarrow$ wkop(p) = p-n dagoen while-kopurua.

Definizio induktiboa:

wkop(x:=t) = 0

wkop(p_1;p_2) = wkop(<u>if</u> b <u>then</u> p_1 <u>else</u> p_2) = wkop(p_1)+wkop(p_2)

wkop(<u>while</u> b <u>do</u> p) = 1+wkop(p)

6.5. *adibidea*: Be*ste funtzio bat while-programen gainean:*

wkab: $WP \rightarrow .^{\cdot}$

$p \rightarrow$ wkab(p) = p-n dauden while kabiatuen kopuru maximoa.

Honela defini daiteke:

wkab(x:=t) = 0

wkab(p_1;p_2) = wkab(<u>if</u> b <u>then</u> p_1 <u>else</u> p_2) = max {wkab(p_1) , wkab(p_2)}

wkab(<u>while</u> b <u>do</u> p) = 1+wkab(p)

6.6. *adibidea*: T^* *sekuentzi multzoaren gainean:*

luz: $T^* \rightarrow .^{\cdot}$

$s \rightarrow$ luz(s) = s-ren luzera,

Indukzioz definituta:

luz(< >) = 0

luz(erantsi(s,t)) = luz(s)+1.

<u>Indukzio estrukturalaren printzipioa</u>: Izan bedi A indukzioz definitutako multzo infinitua, non B oinarrizko multzoa eta K eraikitzaileen multzoa den. A-ko edozein a-k P propietatea betetzen duela frogatzeko, hots, $\forall a\big(a \in A \rightarrow P(a)\big)$ frogatzeko, nahikoa da bi frogapen-pauso hauek burutzea:

a) <u>Oinarrizko pausoa</u>: $\forall a\big(a \in B \rightarrow P(a)\big)$
b) <u>Indukzio-pausoa</u>: Edozein r $\in$ K-rentzat eta edozein $a_1,...,a_n \in$ A-rentzat, non $P(a_1) \wedge ... \wedge P(a_n)$ betetzen den (indukzio-hipotesia): P(r(...,a_1,...,a_n,...)).

6.2. ALGORITMO ERREKURTSIBO ZUZENEN DISEINUA

Ikusi dugun indukzioa da programazioan errekurtsibitate izenez ezagutzen den teknikaren oinarria. Problema baten ebazpidea indukzioz azaldu daitekeenean, zentzuzkoa da ebazpide horretaz baliatzea algoritmoaren diseinuan. Horrela diseinatutako algoritmoari errekurtsibo deitzen zaio, bere buruari deitzen bait dio.

Esate baterako, faktorialaren definizio induktiboa emanda:

$0!=1$ eta $n!=n*(n-1)!$ baldin $n\geq 1$

honako algoritmo errekurtsiboa eraiki daiteke:

```
algoritmoa      FAKTORIAL (n:osoa)
 begin
        if n=0 then itzuli 1
               else itzuli n*FAKTORIAL(n-1)
 end
```

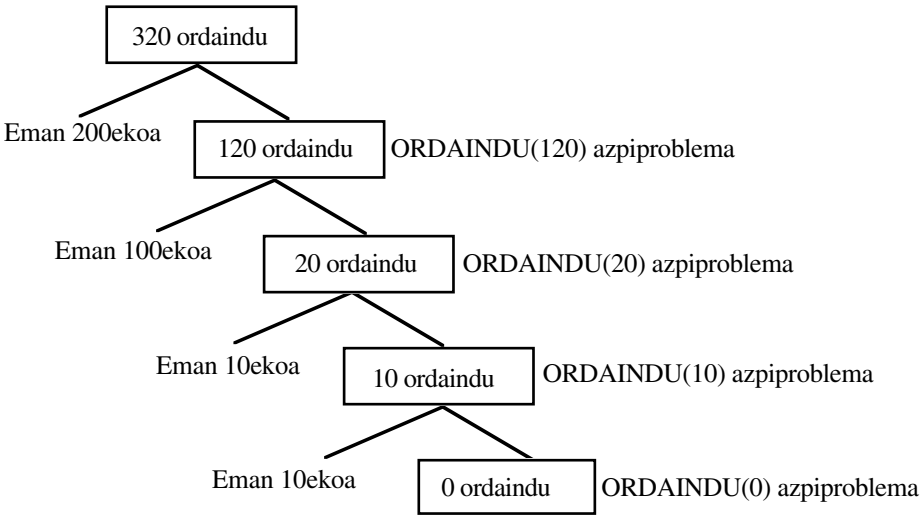
Algoritmo errekurtsiboak problema bat ebazten duenean, problema beraren soluzioak hartzen ditu oinarritzat. Soluzio horiek datu txikiagoetarako (ondo oinarritutako ordena baten araberako txikitasunaz ari gara) emandakoak dira, baina, edonola ere, izaera bereko problema "txikiagoen" soluzioak dira. Aldaketa kuantitatiboa dago ebatzitako problemen artean, ez kualitatiboa. Horregatik, jatorrizko problemaren eta ebazpenean oinarritzat hartzen direnen artean bereizketarik egin nahi denean, azken hauei azpiproblema deitu ohi zaie.

Izaera bereko azpiproblematan deskonposatuz ebatz daitezkeen problemak benetan egokiak dira errekurtsibitateaz baliatuta lantzeko. Horrela eginez gero, soluzio erraz eta argiak erdiesten dira.

Azter dezagun, adibide gisa, beste problema hau:

Pentsa dezagun 500, 200, 100, 50, 25, 5 eta 1eko txanponak nahi adina ditugula, kopuru infinitua beraz, eta $KOP\geq 0$ kopuru zehatza ordaindu beharrean aurkitzen garela. Gainera, zenbat eta txanpon gutxiagorekin ordaindu, orduan eta hobeto. Nola jakin genezake, KOP emanda, zeintzu diren ordainketan erabiliko diren txanponak?

Gure eguneroko jokabidea hauxe litzateke: eskua poltsikoan sartu, gorderik dauzkagun txanponak ikusi, ordaindu beharreko kopurua baino txikiagoa den txanponik handiena hartu eta, behin txanpon hori emanda, oraindik faltan geratzen zaiguna ordaintzeko berdin jokatuko genuke ostera. Ez al du jokabide honek azpiproblematako deskonposaketaren antzik? Adibide honetan garbi ikusten da baduela bai:



Aurkezpen honetan garbi asko antzeman daiteke problemaren soluzioa indukzioz defini daitekeela edota algoritmo errekurtsibo batez. Era honetako algoritmo bat diseinatzeko garrantzizkoa da puntu hauei erreparatzea:

1.- Problemaren eta azpiproblemen arteko aldea, azken batean, ordaindu beharreko kopurua da. Azpiproblematan deskonposatzeko, hortaz, parametro honetan (KOP-en) oinarrituko gara.

Parametroa: KOP≥0

Algoritmoa: ORDAINDU(KOP) = $<k_1, k_2, \ldots, k_n>$ txanpon-sekuentzia itzultzen du non:

$n \geq 0$,

$k_1+k_2+\ldots+k_n=KOP$ eta

$k_1, k_2, \ldots, k_n \in K=\{500, 200, 100, 50, 25, 5, 1\}$

2.- Noiz da soluzioa berehalakoa?
Nolakoa da soluzioa kasu hauetan?

Ordaindu beharreko kopurua zero denean emaitza berehalakoa izango da eta, bistan denez, itzuli beharrekoa sekuentzia hutsa izango da:

. KOP=0 → itzuli < >

3.- Noiz eta nola deskonposatzen da problema azpiproblematan?

Bada, ordaindu beharreko kopurua zero baino hertsiki handiagoa denean, lehenbizi ahalik eta txanponik handiena aukeratu beharko da ordainketarako eta gero ordaindu gabe

geratzen den kopuruaren ordainketaz arduratu beharko dugu, hau da, izaera bereko azpiproblema batez:

```
. KOP>0 →   begin
              x:=max{b∈K / b≤KOP}
              itzuli <x>•ORDAINDU(KOP-x)
            end
```

Egindako problemaren azterketak algoritmo honetara ekarri gaitu:

```
algoritmoa ORDAINDU (KOP:osoa);
 begin {KOP≥0}
      if KOP=0 then itzuli < >
         else      {KOP>0}
                   begin
                    x:=max{b∈K / b≤KOP}
                    itzuli <x>•ORDAINDU(KOP-x)
                   end
 end {irteera=<k1,...,kn> ∧ k1,...,kn∈K     ∧
k1+k2+...+kn=KOP}
```

Diseinua behar bezala bukatzeko, planteatu dugun algoritmoa zuzena dela frogatu beharko genuke, hau da, egiaz problema ebazten duela, batetik, eta bere buruari ez diola etenik gabe deitzen, bestetik. Horrez gain, programazio-lengoaia batean inplementatu beharko litzateke eta iharduketa honek derrigorrez eskatzen du, hala multzoaren eta emaitz sekuentziaren errepresentaziorako datu-motak finkatzea, nola ekintza abstraktuen inplementazioa burutzea: *itzuli, x:=max{b∈K | b≤KOP}*

Ikusi berri ditugun urratsok dira algoritmo errekurtsibo zuzenen diseinuan nabarmenduko ditugun sei etapak:

1.-Parametrizazioa:

Alde batetik parametroak finkatu behar dira, beti ere presente edukita parametro horien arabera definitu behar dela problema errekurtsiboki. Bestetik algoritmoaren zeregina argitu behar da, parametroetan oinarrituta, noski.

Pauso hau funtsezkoa da, parametroen aukeraketak erabat baldintzatzen bait du aurreragoko diseinua. Zenbaitetan, parametro egokiak aukeratu ez eta azpiproblematan deskonposatzea ezinezkoa gertatzen delako edo, atzera jo eta parametro desberdinak hartzera behartuta egongo gara. Hortik atera kontuak. Algoritmoak zer egiten duen xehetasunez definitzeak badu garrantzirik, batez ere deskonposaketa burutzean azpiproblema bakoitzaren esanahia zehatza eta anbiguotasunik gabea izan dadin.

Parametroen gaineko murriztapenak (aurreko baldintzak) eta hasierako deia ere parametrizazioan finkatzen dira. Hasierako deia, jatorrizko problemaren ebazpenerako egiten dena da.

2.- Kasu nabarien azterketa:

Pauso honetan erabakitzen da parametroen zein baliotarako problema esplizituki (ez errekurtsiboki) ebazten den eta zein diren kasu bakoitzari egokitzen zaizkion soluzioak.
Algoritmo errekurtsiboak gutxien-gutxienik kasu nabari bat behar du, bestela ez bait litzateke geldituko. Azken finean, definizio induktiboetan beste horrenbeste gertatzen da: funtzio baten kasu nabaririk gabeko definizio induktiboa indefinituta geratzen da parametroen edozein baliotarako.

3.- Kasu orokorren azterketa:

Parametroen zein baliok eskatzen du azpiproblematako deskonposaketaren bidezko ebazpidea? Galdera horri erantzundakoan kasu desberdinetarako soluzioak finkatu behar dira. Algoritmoari egindako dei errekurtsiboek bat etorri behar dute parametrizazioan finkatutako parametroekin. Horrez gain, lehenengo urratsean definitutako algoritmoaren esanahi generikoak, eta ez besterik, izan behar du urrats honetan deskribatzen diren deien esanahia, bakarrik horrela segurta bait daiteke hasierako problemaren ebazpen zuzena.

4.- Algoritmoaren formulazioa:

Algoritmoak txukuntzea, trinkotzea, borobiltzea dugu helburu pauso honetan. Batzuetan banaka aztertutako kasuak berdin ebazten direla eta, batean bil daitezke. Beste batzuetan, planteatu den ebazpidea dela eta, kasu nabariren bat sekula ez dela gertatuko edota kasu orokorren baten partikularizazioa baizik ez dela konturatuko gara. Zenbaitetan, kasuen azterketa era banatuan egiteak kalkuluen alferrikako errepikapena ekartzen du. Era horretako aspektuak dira algoritmoa formulatzean landu behar direnak, kasukako deskonposaketari algoritmo itxura sendoagoa emateko.

5.- Zuzentasunaren azterketa:

Formulatutako algoritmoak problema egiaz ebazten duela eta, gainera, algoritmoa errekurtsiboa izaki, bere buruari dei-kopuru finitua egiten diola frogatu behar da, bestela ez bait litzateke sekula bukatuko.
Bai bukaera, bai espezifikazioa betetzen duela frogatzeko egokia da indukzioa.

6.-Inplementazioa:

Pauso honetan datu-egiturak eta ekintza abstraktuen finketak zehaztatzen dira. Inplementazioan, errekurtsibitatea onartzen duen lengoaia batean idatzitako azpiprograma errekurtsiboa lortzen da.

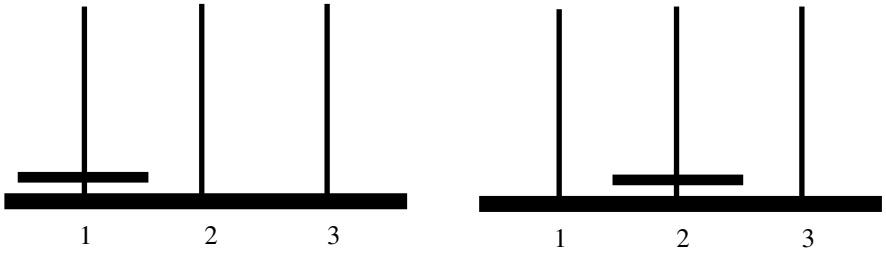
6.7. *adibidea*: *HANOIKO DORREAK: Hiru tantai (1, 2, 3) eta n≥1 tamaina desberdineko diskoak ditugula, 1 tantaian dauden diskoak 2ra pasa nahi dira. Hasieran diskoak handienetik txikienera daude kokatuta 1 tantaian eta 2ra pasatakoan ere ordena berean utzi nahi dira. Gainera mugitzean badago bete beharreko zenbait murriztapen:*

1.- Pausoero disko bakarra mugitzen da tantai batetik bestera.

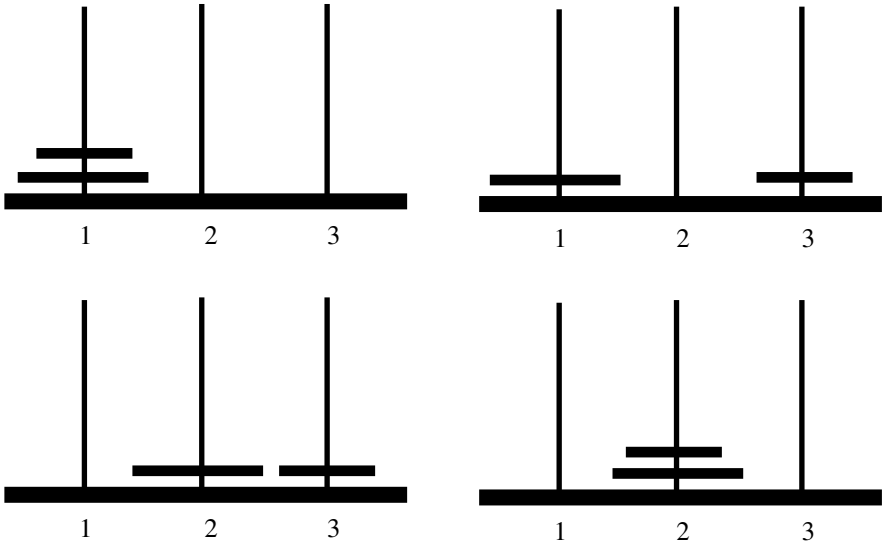
2.- Ezin daiteke disko bat beste txikiago baten gainean kokatu.

3. tantaia laguntzaile gisa erabil daitekeela, asmatu zeregin hau burutuko duen algoritmoa.

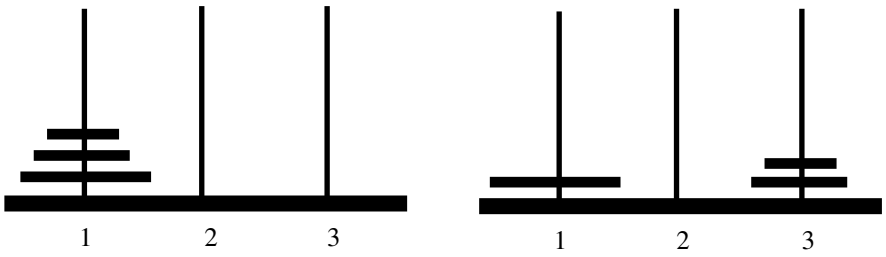
Hasteko problemaren azterketari ekingo diogu. n=1 hartuz gero ez dago inolako arazorik:



n=2 hartuta, 3. dorrearen laguntzaz burutu behar da ekintza:

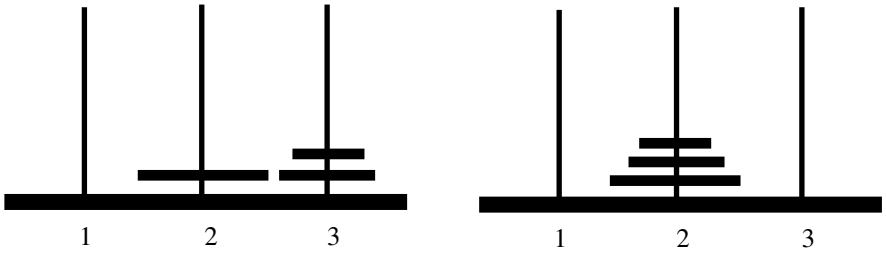


n=3 dela, pausoz pausoko ebazpidea ez da hain nabaria. Hala ere, garbi dago bi disko txikienak tantai laguntzailean utzi behar direla disko handia helburu-tantaira mugituko bada:



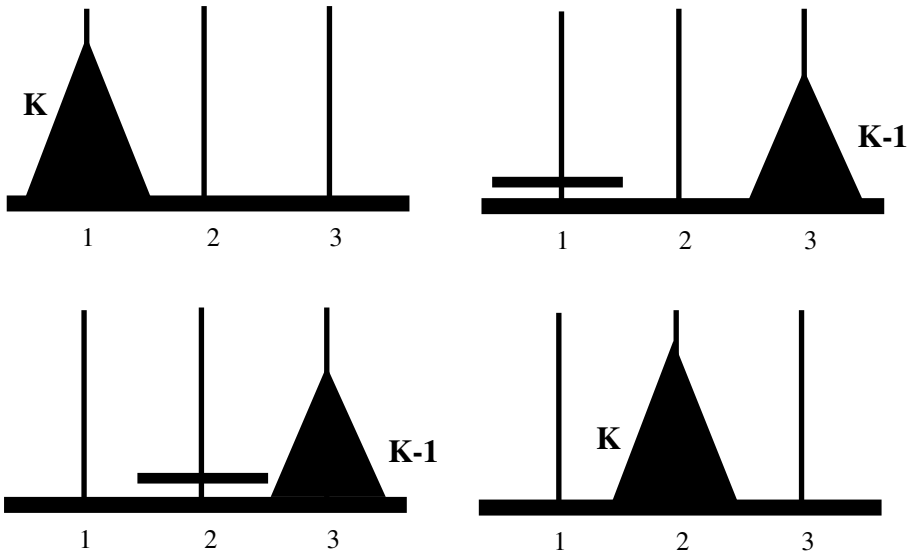
Azpiproblema: Bi disko mugitzea 1etik 3ra, laguntzaile gisa 2a erabiliz.

Ondoren, disko handia 2ra mugitu eta bukatzeko bi txikienak handia dagoen lekura:



Azpiproblema: Bi disko mugitzea 3tik 2ra, laguntzaile gisa 1a erabiliz.

Eta 3 diskotarako erabili den prozedura honek balio dezake edozein k disko-kopururako ere:



k=1 kasua izan ezik, beste gainontzeko guztiak, k=2 barne, metodo honen bidez ebatz daitezke.

Egindako problemaren azterketak diseinu errekurtsiborako bidea ematen du:

- Parametrizazioa:

  k = disko-kopurua (osoa), non k≥1;

  x, y, z: tantaiak

  HANOI(k,x,y,z): x tantaian dauden goiko k diskoak y tantaira mugitzen ditu, z tantaiaren laguntzaz eta arauak errespetatuz.

  Hasierako deia: HANOI(n,1,2,3)

- Kasu nabarien azterketa: Dagoen bakarra k=1 denekoa da. Honelakoetan nahikoa da *mugitu_xtik-yra* ekintza burutzea.

- Kasu orokorren azterketa: k>1 denean. Eginbeharrekoa azpiproblematan banatuz adieraz daiteke:

```
begin
        HANOI(k-1,x,z,y);
        mugitu_xtik_yra;
        HANOI(k-1,z,y,x);
end
```

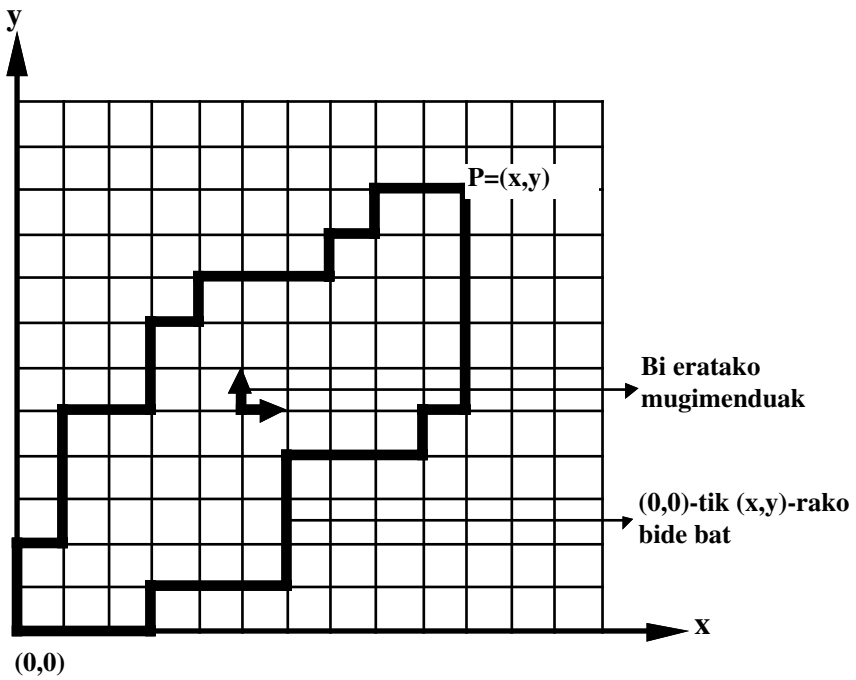
- Algoritmoaren formulazioa:

```
algoritmoa HANOI (k:osoa; x,y,z: tantaiak)
 begin {k≥1 ∧ x tantaian k edo k baino disko gehiago dago }
  if k=1 then mugitu_xtik_yra
  else        begin
                HANOI(k-1,x,z,y);
                mugitu_xtik_yra;
                HANOI(k-1,z,y,x)
              end
 end    { hasieran x tantaian zeuden goiko k diskoak y tantaian daude }
```

- Zuzentasunaren azterketa: Puntu hau oraindik jorratu ez badugu ere, esan dezagun k-ren gaineko indukzioaz frogatu daitekeela HANOI algoritmoaren bukaera eta zuzentasun partziala. Oinarrizko pausoa (k=0 denekoa) agerikoa da. Indukzio-pausorako k-1≥0 balioarekin gertatzen dena hartu behar da indukzio-hipotesitzat, eta ondoren k balioak betetzen dituen propietateak egiaztatu.

- Inplementazioa: Zein datu-mota erabiliko da "tantaiak" adierazteko? Nola aurkeztuko dira emaitzak? *mugitu_xtik_yra* ekintza nola inplementatuko da? Galdera hauei erantzun behar zaie algoritmo hau edozein programazio-lengoaiatan inplementatzean.

6.8. *adibidea*: *Jo dezagun ondoko planoa daukagula eta plano horretako puntu batean egonda pausoak bi zentzutan bakarrik eman daitezkeela, edo gora, edo eskuinera, beti ere aldameneko puntura.*



- *P=(x,y) puntua emanda, diseinatu algoritmo errekurtsiboa O-tik P-ra dauden bide besberdin guztiak kontatzeko.*

Bi era desberdin leudeke problema horri era errekurtsiboan aurre egiteko.

Jo dezakegu, batetik, edozein bide estraineko pausoaz eta ondorengo beste bide batez osatzen dela, eta ideia horretan oinarrituta problemaren adierazpen errekurrentea honela geratuko litzaiguke:

$$BIDEAK(O,P) = BIDEAK(P_1,P) + BIDEAK(P_2,P)$$

"Puntu" motako bi parametro erabiltzen ditugu ebazpide honetan:

$$BIDEAK((x_0,y_0),(x,y)) = BIDEAK((x_0,y_0+1),(x,y)) + BIDEAK((x_0+1,y_0),(x,y))$$

Aitzitik, litekeena da baita ere, edozein bide hasierako azpibide batez eta azkeneko pausoaz osatzen dela suposatzea. Oraingoan adierazpen errekurrenteak itxura hau hartuko du:

$$BIDEAK(O,P) = BIDEAK(O,P_1) + BIDEAK(O,P_2)$$

eta lehenengo parametroa konstantea denez, jarri gabe utz dezakegu:

$$BIDEAK(P) = BIDEAK(P_1) + BIDEAK(P_2)$$

Azken ideia honen araberako diseinua burutuko dugu, hobe bait da ahal den neurrian parametroak aurreztea:

- Parametrizazioa:

  Erabiliko den parametro bakarra P puntua izango da; puntu honen koordenatuak x,y≥0 direla.

  BIDEAK(x,y)= (0,0) puntutik (x,y) puntura dauden bide desberdinen kopurua kalkulatzen du.

- Kasu nabarien azterketa:

  Hiru dira gerta daitezkeen kasu nabariak:

  $x=0 \wedge y=0 \Rightarrow$ *itzuli* 0

  $x=0 \wedge y>0 \Rightarrow$ *itzuli* 1 (behetik gorako bidea)

  $x>0 \wedge y=0 \Rightarrow$ *itzuli* 1 (ezkerretik eskuinerako bidea)

- Kasu orokorren azterketa:

  Kasu orokor bakarra gerta daiteke:

  $x>0 \wedge y>0 \Rightarrow$ *itzuli* BIDEAK(x-1,y) + BIDEAK(x,y-1)

- Algoritmoaren formulazioa:

```
algoritmoa BIDEAK(x,y:osoa);
begin {x,y≥0}
   if (x=0 and y=0) then itzuli 0
   else if (x=0 or y=0) then itzuli 1
   else itzuli BIDEAK(x-1,y) + BIDEAK(x,y-1)
end    {(0,0) puntutik (x,y) puntura dauden bide desberdinen kopurua itzuli
        du}
```

- *P=(x,y) puntua emanda, diseinatu algoritmo errekurtsiboa O-tik P-ra dauden bide besberdin guztiak sortzeko.*

Berriro ere bidea bi eratara uler daiteke. Pauso batez bukatzen den puntu segidatzat hartzen bada, diseinua modu honetan gauzatuko litzateke:

- Parametrizazioa:

  x,y≥0.

  BIDEAK(x,y)= (0,0) puntutik (x,y) puntura joateko dauden bide desberdin guztiak sortzen ditu.

- Kasu nabarien azterketa:

  x=0 ∧ y=0 ⇒ *itzuli* <> (bide hutsa)
  x=0 ∧ y>0 ⇒ *itzuli* < (0,1), (0,2), ..., (0,y) >
  x>0 ∧ y=0 ⇒ *itzuli* < (1,0), (2,0), ..., (x,0) >

- Kasu orokorren azterketa:

```
x>0 ∧ y>0 ⇒ begin
                BIDEAK(x-1,y); erantsi_guzti_hauei(x,y);
                BIDEAK(x,y-1); erantsi_guzti_hauei(x,y);
            end
```

Baina problemaren ebazpena zehaztasun gehiagoz adierazteko aldagai berri bat erabiltzea komeni zaigu, sortutako bideak bertan gordetzeko. Aldaketa hau hasiera-hasieratik, parametrizaziotik, egin behar da. Beraz, nahiz eta kasu orokorren azterketara iritsiak ginen, atzera jo eta diseinuari berrekin egingo diogu.

- Parametrizazioa:

  x,y≥0.

  BIDEAK(x,y)= (0,0) puntutik (x,y) puntura joateko dauden bide desberdin guztiak itzultzen ditu BIDESEK-en. Bidea puntu-sekuentzia izango da eta BIDESEK bide-sekuentzia, beraz.

- Kasu nabarien azterketa:

  x=0 ∧ y=0 ⇒ BIDESEK:=<>
  x=0 ∧ y>0 ⇒ BIDESEK:=< (0,1), (0,2), ..., (0,y) >
  x>0 ∧ y=0 ⇒ BIDESEK:=< (1,0), (2,0), ..., (x,0) >

- Kasu orokorren azterketa:

```
x>0 ∧ y>0 ⇒ begin
              BIDEAK(x-1,y,BIDESEK1); erantsi_guztiei(BIDESEK1,x,y);
              BIDEAK(x,y-1,BIDESEK2); erantsi_guztiei(BIDESEK2,x,y);
              BIDESEK:=BIDESEK1•BIDESEK2;
              end
```

- Formulazioa:

```
algoritmoa BIDE (x,y:osoa; BIDESEK:bidesekuentzia);
  begin {x,y≥0}
        if (x=0 and y=0) then BIDESEK:=<>
        else if x=0 then {x=0 ∧ y>0}
                    BIDESEK:=< (0,1), (0,2), ..., (0,y) >
        else if y=0 then  {x>0 ∧ y=0}
                    BIDESEK:=< (1,0), (2,0), ..., (x,0) >
             else begin {x>0 ∧ y>0}
                  BIDEAK(x-1,y,BIDESEK1); erantsi_guztiei(BIDESEK1,x,y);
                  BIDEAK(x,y-1,BIDESEK2); erantsi_guztiei(BIDESEK2,x,y);
                  BIDESEK:=BIDESEK1•BIDESEK2;
                end
  end {BIDESEK= 0-tik (x,y) punturako bide guztien sekuentzia}
```

Algoritmo errekurtsiboen zuzentasun-azterketa

Algoritmo errekurtsiboa zuzena dela frogatzeko bi atal jorratu behar dira:

a) Espezifikazioa betetzen duela, nahi duguna egiten duela, alegia.

b) Bukatzen dela, hots, bere buruari aldi-kopuru finituan deitzen diola. Bata nahiz besterako tresnarik erabilgarriena indukzioa da.

Nahiz eta zuzentasunaren azterketa boskarren etapan burutzen den, lehenagoko etapa guztietan presente eduki behar da nondik nora gauzatuko den frogapena. Egindako diseinu-pausoen ezaugarriek horretara bultzatu nahi dute.

Aipatu ditugun frogapen-ataletatik bigarrengoari ekingo diogu lehenbizi bukaera-frogapena argiago edo, behintzat, errazagoa gertatu ohi bait da. Iterazioen bukaera landu genuenean ondo oinarritutako ordenak hartu genituen azpieredutzat, algoritmo errekurtsiboen bukaera, ordea, egokiago aztertzen da indukzioz.

Erne eta taxuz programatu ezean oso erraza da algoritmo errekurtsiboetan akats txikiren bat egin eta algoritmoak bere buruari etengabe deitzea. Adibidez, ondoko algoritmo errekurtsiboa ez da amaituko, zerorako ezik, parametroen beste edozein baliotarako:

```
algoritmoa IDENT (a:osoa);
 begin {a≥0}
        if a=0 then itzuli 0
        else if odd(a) then itzuli IDENT(a-2)+2
        else itzuli IDENT(a-1)+1
 end {IDENT(a)=a}
```

Goazen, bada, IDENT(a)-k dei-kopuru finitua sortzen duela frogatzera.

Zenbaki arruntetako indukzioz frogatuko dugu edo, zehatzago esanda, a parametroaren gaineko indukzioaz. Gainera, algoritmoaren ezaugarriek indukzio osotua erabiltzea eskatzen digute.

(1) Oinarrizko pausoa:

IDENT(0)-k 0 dei sortzen du $\Rightarrow$ IDENT(0)-k dei-kopuru finitua sortzen du.

(2) Indukzio-pausoa:

Edozein a>0 hartuta, IDENT(a)-k dei-kopuru finitua sortzen duela frogatu nahi dugu.

Indukzio-hipotesi gisa: $\forall k(0 \leq k < a \rightarrow$ IDENT(k)-k dei-kopuru finitua sortzen du)

Bi kasu bereiziko ditugu, dei errekurtsiboetan parametroaren eguneraketa bi bide desberdinetatik egiten bait da:

a.- a>0 bikoitia denean:

a>0 ∧ a bikoitia → 0≤a-1<a →

IDENT(a-1)-ek dei-kopuru finitua sortzen du →

IDENT(a)-k IDENT(a-1)-ek baino dei bat gehiago sortzen du →

IDENT(a)-k dei-kopuru finitua sortzen du.

b.- a>0 bakoitia denean:

-1≤a-2<a beteko dela zierto badakigu, baina ez 0≤a-2<a; eta horrek esan nahi du a-2 baliorako ezin dela indukzio-hipotesia ontzat hartu.

Ezin da, beraz, IDENT(a–2)-k dei-kopuru finitua sortzen duenik ziurtatu eta, ondorioz, ezta IDENT(a)-k frogatu nahi genuena betetzen duenik ere.

Adibide horretan ikusitakoaren arabera indukzio estrukturala tresna egokitzat jo behar da algoritmo batek bere buruari aldi-kopuru finituan deitzen diola frogatzeko.

Lehenxeago ikusitako ORDAINDU algoritmoa aztertuko dugu ondoren:

6.9. *adibidea*: *Frogatu propietate hau:*

∀p(p≥0 → ORDAINDU(p)-k dei-kopuru finitua sortzen du)

Kasu honetan ere .˙·˙-ren gaineko indukzio estrukturala erabiliko dugu, eta indukzio osotua gainera, izan ere ORDAINDU(p)-k ORDAINDU(p-x)-ri egiten bait dio dei, non x≤p den eta x∈K={500, …, 1}.

(1) ORDAINDU(0)-k 0 dei sortzen du → ORDAINDU(0)-k dei-kopuru finitua sortzen du

(2) p>0 Indukzio-hipotesia:

∀k(0≤k<p → ORDAINDU(k)-k dei-kopuru finitua sortzen du)

ORDAINDU(p)-k ORDAINDU(p-x)-k baino dei bat gehiago sortzen du, non x≤p eta x∈K diren, eta 0≤p-x<p denez, indukzio-hipotesiaz: ORDAINDU(p-x)-k dei-kopuru finitua sortzen du; beraz, ORDAINDU(p)-k ere dei-kopuru finitua sortzen du.

Adibide horretan ez da ez samurra dei-kopurua zehatz-mehatz finkatuko duen p-ren araberako adierazpena asmatzea, k multzoak ere bai bait du eraginik adierazpen horretan. Hala ere, beste zenbait algoritmotan nahiko erraza gertatzen da adierazpen hori aurkitzea. Holakoetan, dei-kopurua zehaztu, edo gutxienik mugatzen denean, bukaeraren frogapena ezezik algoritmoaren konplexutasunaren neurria ere lortzen da.

Ikus dezagun adibide sinple batez nola kalkula daitekeen dei-kopurua.

6.10. *adibidea*: *FAKT(n)-ren bukaera-frogapena, dei-kopuruaren kalkuluan oinarritua.*

Algoritmo errekurtsiboaren azterketak ondoko ekuazio errekurrentea paratzeko aukera ematen digu:

$$\text{dei\_kop}(\text{FAKT}(n)) = \begin{cases} 0 & \text{baldin } n = 0 \\ 1 + \text{dei\_kop}(\text{FAKT}(n-1)) & \text{baldin } n \geq 1 \end{cases}$$

Ekuazio errekurrente horretan oinarrituta, honako propietate hau frogatuko dugu indukzioz:

$\forall n(n \geq 0 \rightarrow \text{dei-kop}(\text{FAKT}(n))=n)$

Frogapena:

1.- Oinarrizko pausoa:

$n=0 \rightarrow \text{dei-kop}(\text{FAKT}(0))=0$

2.- Indukzio-pausoa: $n \geq 1$

Indukzio-hipotesia: $\forall k(k \geq 0 \rightarrow \text{dei-kop}(\text{FAKT}(k))=k)$

$\text{dei -kop}(\text{FAKT}(n))=1+\text{dei-kop}(\text{FAKT}(n-1))=1+(n-1)=n$

FAKT-en kasuan intuizioz asma daiteke zein den dei-kopuruaren adierazpena algoritmoari edo ekuazio errekurrenteari begiratu besterik gabe. Ez da beti horrela gertatzen, ordea. Ekuazio errekurrenteen hedapenezko ebazpidea erabiltzen da maiz dei-kopuruaren adierazpen ez-errekurrentea lortzeko. Gero, asmatutako adierazpena zuzena dela egiaztatu beharko da.

Ekuazio errekurrenteak hedapenez ebazteko ekuazioaren ezker-parteko funtzioa bere balioaz ordeztu eta adierazpen orokor bat lortzen da, gero definizioaren kasu nabarira partikularitzen da delako adierazpen orokor hori.

6.11. *adibidea*: *HANOI(k, x, y, z) algoritmoaren bukaera-frogapena, dei kopuruaren kalkuluan oinarritua.*

Agerikoa da tantaiek ez dutela eraginik dei-kopuruan. Disko-kopuruaren, k-ren, baldintzapekoa da dei-kopurua. Horregatik HANOI(k) idatziko du hemendik aurrera, nahiz-eta idazkera hori ez datorren bat parametrizazioan finkatutakoarekin. Hasteko, algoritmoari erreparatuta, honako ekuazioa errekurrentea lortzen da:

$$\text{dei\_kop}(\text{HANOI}(k)) = \begin{cases} 0 & \text{baldin } k = 1 \\ 2 + 2 * \text{dei\_kop}(\text{HANOI}(k-1)) & \text{baldin } k > 1 \end{cases}$$

Ekuazio hori ez da FAKT(n)-ren adibidean ikusi duguna bezain sinplea. Kasu honetan, k-ren araberako adierazpen esplizitua lortzeko ekuazio errekurrenteen hedapenaz baliatuko gara:

$$\begin{aligned}
dei_kop(HANOI(k)) &= 2 + 2*dei_kop(HANOI(k-1)) \\
&= 2 + 2*\left(2 + 2*dei_kop(HANOI(k-2))\right) \\
&= 2 + 2^2 + 2^2*dei_kop(HANOI(k-2)) \\
&= 2 + 2^2 + 2^2*\left(2 + 2*dei_kop(HANOI(k-3))\right) \\
&= 2 + 2^2 + 2^3 + 2^3*dei_kop(HANOI(k-3)) \\
&= \ldots
\end{aligned}$$

Orokorrean:

$$dei\_kop(HANOI(k)) = 2 + 2^2 + 2^3 + \ldots + 2^i + 2^i*dei\_kop(HANOI(k-i))$$

Eta k-i=1 deneko kasura mugatzen badugu:

$$\begin{aligned}
dei_kop(HANOI(k)) &= 2^1 + 2^2 + 2^3 + \ldots + 2^i + 2^i*dei_kop(HANOI(1)) \\
&= 2^1 + 2^2 + 2^3 + \ldots + 2^i + 2^i*0 \\
&= 2^1 + 2^2 + 2^3 + \ldots + 2^i
\end{aligned}$$

Kontuan hartuz, gainera, k-i=1 denez, i=k-1 izango dela:

$$dei\_kop(HANOI(k)) = 2^1 + 2^2 + 2^3 + \ldots + 2^{k-1}$$

Kondaira zahar batek dioenez Hanoiko dorreak (64 disko dauzkate) dauden monastegiko fraileek mugimendu bat egiten omen dute egunero eta lan hori bukatutakoan bukatuko omen da mundua.

Aidanean, pertsona batek disko-mugimendu bat egingo balu segundoko, nahiz eta inolako hankasartzerik egin ez, ez omen luke bizitza osoan egin beharrekoa amaituko. Arrazoi ote dute kondaira zahar honetan sinesten dutenek? Bada, kalkula dezagun disko-mugimenduen kopurua k-ren arabera, eta horretara jakingo dugu ea baieztapen hori zentzuzkoa den. Mugimendu-kopurua adierazten duen ekuazio errekurrentea:

$$mug\_kop(HANOI(k)) = \begin{cases} 1 & \text{baldin } k = 1 \\ 1 + 2*mug\_kop(HANOI(k-1)) & \text{baldin } k > 1 \end{cases}$$

eta hedapenez ebatziz gero:

$$\forall k\left(k \geq 1 \rightarrow mug\_kop\left(HANOI(k)\right) = 2^1 + 2^2 + \cdots + 2^{k-1} + 2^k\right)$$

Horretara, k=64 dela joz gero mugimenduen kopurua $7{,}3787*10^{19}$-1 izango da. Eguneko mugimendu bakarra egiten dela $2{,}016*10^{17}$ urte beharko lirateke, eta segundoko mugimendu bat egiten dela, berriz, $2{,}3398*10^{13}$ urte. Ez luke ez, pertsona batek bizitza osoan bukatuko egiteko hori.

Laburbilduz, bi izan dira algoritmo errekurtsiboak dei-kopuru finitua eragiten duela frogatzeko azaldu diren metodoak:

1) "Algoritmoak aldi-kopuru finituan deitzen dio bere buruari" propietatearen indukzio estrukturalaren bidezko frogapena.

2) Dei-kopurua zehaztuko (edo mugatuko) duen adierazpena aurkitzea. Bi bide leudeke horretarako:

a) Adierazpen hori sumatu eta ondoren indukzioz betetzen dela frogatzea.

b) Ekuazio errekurrentetik adierazpen hori lortzea.

Goazen orain, metodoen azalpenak ikusi ondoren, (x,y) punturako bideen problema ebazteko eman ditugun algoritmoen bukaera aztertzera.

6.12. *adibidea*: *Frogatu BIDEAK eta BIDE algoritmoak bukatu egiten direla.*

Bi kasuotan adierazpen errekurrente berbera lor daiteke, hau da, parametro berdinetarako dei-kopurua ez da aldatuko batetik bestera; beraz, algoritmoaren izena erabiltzeko beharrik gabe, hemendik aurrerako azterketa balekoa da bi algoritmoetarako:

$$\text{dei\_kop}(x,y) = \begin{cases} 0 & \text{baldin } x = 0 \ \vee \ y = 0 \\ 2 + \text{dei\_kop}(x-1,y) + \text{dei\_kop}(x,y-1) & \text{baldin } x,y > 0 \end{cases}$$

Ez da hain samurra, baina, ekuazio errekurrente horren ebazpena. Horregatik dei-kopurua finitua dela frogatzera joko dugu zuzenean. Aukeratu dezagun, hasteko, zein argumenturen gainean erabili behar dugun indukzioa: x-ren ala y-ren gainean?

Bada, dei-kopuruaren adierazpen induktibotik erraz erator daitekeenez, bien baturaren gainean.

1) $x+y=0 \Rightarrow x=0 \wedge y=0 \Rightarrow$ dei_kop(x,y)=0 $\Rightarrow$ dei_kop(x,y) finitua da

2) Bi kasu bereizi behar dira:

a) $x=0 \vee y=0 \Rightarrow$ dei_kop(x,y)=0 $\Rightarrow$ dei_kop(x,y) finitua da

b) $x>0 \wedge y>0 \Rightarrow x+y>0 \Rightarrow x+y-1 \geq 0$

Indukzio-hipotesia x+y-1 adierazpenetik hartuta:

dei_kop(x-1,y) finitua da eta dei_kop(x,y-1) finitua da,

ondorioz:

dei_kop(x,y) ere finitua da.

Algoritmo errekurtsiboen egiaztapena

Har dezagun sarrerako algoritmo hau:

```
algoritmoa   FAKT (n:osoa)
    begin    {n≥0}
             if n=0 then itzuli 1
                  else itzuli n*FAKT(n-1)
    end      {FAKT(n)-k n! itzultzen du}
```

eta jo dezagun espezifikazioa betetzen dela egiaztatzera. Kontuan hartuta bukaera dagoeneko frogatuta edukiko dugula, aski da, indukzioaz baliatuta oraingoan ere, n=0 balio nabariarentzat eta, n-1 balioa hartuz betetzen dela suposatuta, n balio orokorrarentzat ere espezifikazioa betetzen dela frogatzea. Beraz, gure oraingo adibidera etorriz, froga dezagun FAKT(n) algoritmoak n! balioa itzultzen duela:

1) $n \geq 0 \wedge n=0 \Rightarrow$ FAKT(n)-k itzulitakoa=1 =0! =n! $\Rightarrow$ FAKT(n)-k n! itzultzen du
2) Indukzio-hipotesia: FAKT(n-1)-ek (n-1)! itzultzen du
   $n \geq 0 \wedge n \neq 0 \Rightarrow$ FAKT(n)-k itzulitakoa=n*FAKT(n-1)
   $\Rightarrow$ FAKT(n)-k itzulitakoa=n*(n-1)! = n!

Hala ere, egiaztapenak, formala izango bada behinipehin, prozedura eta funtzioen arteko bereizketa eskatzen du, prozedura errekurtsiboari eginiko deiaren eta funtzioari eginikoaren artean bai bait dago diferentzia nahikoa nabarmenik, iteratiboak zirenean gertatzen zen bezalaxe.

Oraingoan ere, prozedura- edo funtzio-gorputzak espezifikazioa betetzen duela frogatzen saiatuko gara, hartara edozein deirentzat antzeko zerbait inferitu ahal izateko. Gogoan izan horixe dela prozedura edo funtzio iteratiboetan egiten genuena. Hala ere, eta honetan dago koxka, algoritmo errekurtsiboen gorputzean gutxienez bere buruari egindako dei bat ageri da (bat baino gehiago ere egon daiteke), eta, jakina, dei horiei buruz ezer ez dakigula ezinezkoa da gorputzari buruz taxuzko egiaztapenik egitea. Hara: gorputzaren egiaztapenak deien aurretiko frogapena eskatzen du eta, bestalde, deiak egiaztatuko badira lehenago gorputzek espezifikazioa betetzen dutela frogatu behar da. Nola askatu korapilo hau? Izan dezagun presente algoritmo errekurtsiboak definizio induktiboetatik eratorritako ebazpideak direla eta, gauzak horrela, indukzio estrukturala izango da berriz ere egiaztapena formalki burutzeko aukera eskainiko digun giltza.

Funtzio Errekurtsiboen Egiaztapena (FERRE)

Izan bedi:

function FERR($\bar{x}$:< parametroen_ mota >):< emaitzaren_ mota >; S;

Honako hau da formula berri honen muina: funtzioaren gorputzak espezifikazioa betetzen duela frogatzen bada edozein deik hala egiten duela hipotesitzat hartuta, orduan funtzioak beteko du espezifikazioa.

Indukzio-hipotesia edozein deirentzat hartzen bada ere, frogapenean Sn ageri diren deietarako bakarrik erabiltzen da. Horren arabera, bistan da bukaera frogatuz gero dei horiek gero eta parametroen balio txikiagoak hartuko dituztela, eta indukzioaren erabilera akatsik gabea egingo da, beraz.

6.13. *adibidea*: *Frogatu ondoko funtzioaren zuzentasuna:*

```
function FAKT (n:integer): integer;
  begin    {n≥0}
           if n=0 then FAKT:=1
                  else FAKT:=n*FAKT(n-1)
  end      {FAKT=n! }
```

Badakigu dagoeneko FAKT(n)-k sortzen duen dei-kopurua n dela. Froga dezagun orain espezifikazioa betetzen duela:

1.- $(n \geq 0 \wedge n = 0) \rightarrow \{n = 0\}$ FAKT:= 1 $\{n = 0 \wedge FAKT = 1\} \rightarrow$ FAKT = n!

2.- Indukzio_hipotesia: $\forall n(n \geq 0 \rightarrow FAKT(n) = n!)$, bereziki $FAKT(n-1) = (n-1)!$

3.- $n \geq 0 \wedge n \neq 0 \rightarrow n > 0 \rightarrow n - 1 \geq 0 \rightarrow FAKT(n-1) = (n-1)! \rightarrow$

$\{n * FAKT(n-1) = n!\}$ FAKT:= $n * FAKT(n-1)$ $\{FAKT = n!\}$ (2)

4.- $\{n \geq 0\}$ begin $\cdots$ end $\{FAKT = n!\}$ 1, 3, (BDE)

5.- $\forall n(n \geq 0 \rightarrow FAKT(n) = n!)$ 4, (FERRE)

Prozedura Errekurtsiboen Egiaztapena (PERRE)

Prozedura errekurtsiboetan ere $\{\phi\}$ S $\{\psi\}$ hirukotearen frogapena aurreko deiak espezifikazio horretara egokitzen direla indukzio-hipotesitzat hartuz burutzen da. Noski, horretarako sortzen diren dei errekurtsiboen kopurua finitua dela jakin behar da, bestela indukzioaren erabilera zalantzazkoa bait da.

Eredu honetako espezifikazioa emanda:

$$\underline{\text{procedure}}\ \text{PERRE}\left(\bar{x}: < \text{parametroen\_mota} >\right);\ \text{S};$$

egiaztapen-araua, bereizte-baldintza eta guzti, hau izango da:

6.14. ***adibidea*****:** *Frogatu ondoko prozeduraren zuzentasuna:*

```
procedure FAKT (n:integer; F:integer);
  begin   {n≥0}
          if n=0 then F:=1
                 else begin
                        FAKT(n-1,F);
                        F:=n*F
                      end
  end     {F=n! }
```

FAKT(n)-k n dei sortzen duela badakigula, froga dezagun orain espezifikazioa betetzen duela:

1.- $(n \geq 0 \wedge n = 0) \rightarrow \{n = 0\}\ F := 1\ \{n = 0 \wedge F = 1\} \rightarrow\ F = n!$

2.- Indukzio_hipotesia: $\forall n \forall F\left(\text{bereiz}\ (n, F) \rightarrow \{n \geq 0\}\text{FAKT}(n, F)\{F = n!\}\right)$

3.- $\text{bereiz}(n-1, F)$

4.- $\{n-1 \geq 0\}\ \text{FAKT}(n-1, F)\ \{F = (n-1)!\}$ 2, 3

5.- $(n \geq 0 \wedge n \neq 0) \rightarrow \{n-1 \geq 0\}\ \text{FAKT}(n-1, F)\{F = (n-1)!\} \rightarrow$
$\{n * F = n!\}\ F := n * F\{F = n!\}$

6.- $\{n \geq 0\}\ S\ \{F = n!\}$ 1, 5, (BDE)

6.15. *adibidea*: *x≥0 eta y≥0 zenbaki arrunten arteko zatidura eta hondarra kalkulatuko duen prozedura errekurtsiboa diseinatu. Prozedura horretan ezin izango da batuketaz eta kenketaz besterik erabili:*

Diseinuaren oinarritzat bi ondoko definizio induktiboak hartuko dira:

$$\text{hondar}(x,y) = \begin{cases} x & \text{baldin } x < y \\ \text{hondar}(x-y,y) & \text{baldin } x \geq y \end{cases}$$

$$\text{zatidura}(x,y) = \begin{cases} 0 & \text{baldin } x < y \\ 1 + \text{zatidura}(x-y,y) & \text{baldin } x \geq y \end{cases}$$

- Parametrizazioa: Diseinu errekurtsiboa x parametroa (zatikizuna) aldatuz bideratzen da, baina y ere (zatitzailea) kontuan hartu behar da emaitza lortzean:
  x,y: zatikizuna eta zatitzailea, hurrenez-hurren; non $x \geq 0 \wedge y \geq 0$.
  ZAT(x,y,k,h)= x eta y-ren arteko zatidura k-n uzten du eta hondarra h-n.

- Kasu nabarien azterketa:
  $x<y \Rightarrow k=0 \wedge h=x$
  $x=y \Rightarrow k=1 \wedge h=0$

- Kasu orokorren azterketa:

```
x>y ⇒ begin
          ZAT(x-y,y,k',h);
          k:=k'+1;
      end
```

- Algoritmoaren formulazioa:

```
algoritmoa ZAT(x,y,k,h:osoak);
 begin  {x≥0 ∧ y>0 }
  if x<y then   begin
                      h:=x;
                      k:=0
                end
  else if x=y then     begin
                            h:=0;
                            k:=1
                       end
  else begin
                ZAT(x-y,y,k,h);
                k:=k+1
       end
end     {x=y*k+h ∧ h<y }
```

- Zuzentasunaren azterketa:

Indukzio osotuaz erraz froga daiteke ondoko propietatea:

$\forall x \forall y \left(x \geq 0 \wedge y > 0 \rightarrow \text{"ZAT}(x, y, k, h) - k \text{ dei} - \text{kopuru finitua sortzen du"} \right)$

Bukaera frogatzeko bigarren bidea hartuko bagenu, hau da, dei-kopurua zehazki kalkulatzearena:

$$\text{dei\_kop}\left(\text{ZAT}(x,y)\right) = \begin{cases} 0 & \text{baldin } x \leq y \\ 1 + \text{dei\_kop}\left(\text{ZAT}\left(x - y, y\right)\right) & \text{baldin } x > y \end{cases}$$

$$\begin{aligned} \text{dei\_kop(ZAT(x,y))} &= 1+\text{dei\_kop(ZAT(x-y,y))} = \\ &= 1+(1+\text{dei\_kop(ZAT(x-2*y,y))}) = \\ &= 2+\text{dei\_kop(ZAT(x-2*y,y))} = \\ &= 3+\text{dei\_kop(ZAT(x-3*y,y))} \\ &= \ldots \\ &= i+\text{dei\_kop(ZAT(x-i*y,y))} \end{aligned}$$

Kasu nabaria gertatzen denean:

$x - i*y \leq y \rightarrow i*y < x \leq (i+1)*y \rightarrow \text{dei\_kop(ZAT(x,y))} = i+0 = i$

$x - i*y \leq y \rightarrow i*y < x \leq (i+1)*y \rightarrow$

$(x \text{ mod } y = 0 \wedge i+1 = x \text{ div } y) \vee (x \text{ mod } y \neq 0 \wedge i = x \text{ div } y)$

Honenbestez:

$$\text{dei\_kop}\left(\text{ZAT}(x,y)\right) = \begin{cases} x \text{ div } y - 1 & \text{baldin } x \text{ mod } y = 0 \\ x \text{ div } y & \text{baldin } x \text{ mod } y \neq 0 \end{cases}$$

Egiazta dezagun orain espezifikazioa betetzen duela:

$$\left\{x \geq 0 \wedge y > 0\right\} \; S \; \left\{x = y*k + h \wedge h < y\right\}$$

1.- $\left\{x \geq 0 \wedge y > 0 \wedge x < y\right\}$ k:= 0; h:= x $\left\{x \geq 0 \wedge y > 0 \wedge x < y \wedge k = 0 \wedge h = x\right\} \rightarrow$
$\left\{x = y*k + h \wedge h < y\right\}$

2.- $\left\{x \geq 0 \wedge y > 0 \wedge x = y\right\}$ k:= 1; h:= 0 $\left\{x \geq 0 \wedge y > 0 \wedge x = y \wedge k = 1 \wedge h = 0\right\} \rightarrow$
$\left\{x = y*k + h \wedge h < y\right\}$

3.- Indukzio_hipotesia:

$\text{bereiz}(x - y, y, k, h) \rightarrow \left\{x - y \geq 0 \;\wedge\; y > 0\right\} \text{ZAT}\left(x - y, y, k, h\right)$
$\left\{x - y = y*k + h \wedge h < y\right\}$

4.- $\left\{x \geq 0 \;\wedge\; y > 0 \;\wedge\; x > y\right\} \rightarrow \quad \left\{x - y \geq 0 \wedge y > 0\right\}$
$\text{ZAT}\left(x - y, y, k, h\right)$
$\left\{x - y = y*k + h \wedge h < y\right\} \rightarrow \left\{x = y*(k + 1) + h \wedge h < y\right\}$

5.- $\left\{x = y*(k + 1) + h \wedge h < y\right\}$ k:= k + 1 $\left\{x = y*k + h \wedge h < y\right\}$

- Inplementazioa:

Ondoko hau, esate baterako:

```
procedure ZAT(x,y:integer; var k,h:integer);
 begin  {x≥0 ∧ y>0 }
  if x<y then   begin
                        h:=x;
                        k:=0
                end
  else if x=y then      begin
                                h:=0;
                                k:=1
                        end
  else begin
                ZAT(x-y,y,k,h);
                k:=k+1
       end
 end    {x=y*k+h ∧ h<y }
```

6.3. ARGITASUN/ERAGINKORTASUN ERLAZIOA ERREKURTSIOAREN ERABILERAN

Zenbaitetan, emandako problemari aurre egiterakoan ebazpen induktiboa bururatzen zaigunean, errekurtsibitateak ematen digu algoritmo ulerterraz eta labur antzekoak idazteko aukera. Maiz, ordea, ez-eraginkorrak izaten dira algoritmo horiek. Hara zergatik:

1) Algoritmoak dei berri bat sortzen duenean parametro eta aldagai lokal guztien balioak memorian gordetzen dira, sortutako deiaren (zeinek era berean beste asko sor ditzakeen) exekuzioa bukatzen denean balio horiek berreskuratzeko. Gordeketa horiek memoria kontsumitzen dute noski eta, gainera, gordeketatan eta bereskuraketatan denbora ere joaten da.

Har dezagun BIDE (x_0,y_0,x,y,BIDESEK) algoritmoa. Kontuan hartzen badugu, batetik, BIDESEK bide-sekuentzia bat dela, hau da, bikote-sekuentzien sekuentzia, bestetik, parametro hori beste gainerako laurekin gorde egiten dela dei berria sortu orduko, eta gainera dei bakoitzak beste bi sortzen dituela, exekuzioa neketsua izango dela ondoriozta dezakegu.

Badira, esandakoaz gain, gordeketak eta berreskuraketak beharrezkoak ez direneko kasuak. Esate baterako, FAKT algoritmoan edozein gordeketaren 'ken 1' egin genezakeen, eta berreskuratu beharrean 'gehi 1'.

2) Algoritmoaren konputazioak behin baino gehiagotan sortzen badu dei berdina, azken finean kalkulu bera errepikatu egingo da dei hori gertatzen den adina aldiz. Adibide garbia dugu ondoko hau, Fibonacci-ren segidaren n-garren gaia kalkulatzen duen algoritmo errekurtsiboa:

```
function Fib(n:integer): integer:
  begin  {n≥0}
        if (n=0) or (n=1) then  Fib:=n
        else Fib:=Fib(n-1)+Fib(n-2)
  end  {Fib=fibonacci-ren segidaren n-garren gaia}
```

Fib (4) kalkulatzeko:

$$\begin{aligned} Fib(4) &= Fib(3) + Fib(2) = \\ &= (Fib(2) + Fib(1)) + Fib(2) = \\ &= Fib(1) + Fib(0) + Fib(1) + Fib(1) + Fib(0), \end{aligned}$$

eta jakina, zenbat eta handiago izan n, orduan eta konputazioen errepikapen nabariagoa. Edozein algoritmo iteratibok Fibonacciren segidaren n-garren gaia azkarrago kalkulatuko luke.

Beste zenbait kalkulu-errepikapen algoritmoari ukituak eginez konpon daiteke. Era honetako eskemetan esate baterako:

```
function F(x̄: < parametroen_mota >) : < emaitzaren_mota >;
 begin
       if bal1(x̄) then F:= em(x̄)
       else if bal2(F(t1(x̄)), F(t2(x̄))) then F:= g(F(t1(x̄)), F(t2(x̄)))
       else F:= h(F(t1(x̄)), F(t2(x̄)))
 end
```

aldaketa hauek egin daitezke exekuzioa hobetze aldera:

```
function F(x: < parametroen_mota >) : < emaitzaren_mota >;
 begin
       if bal1(x) then F:= em(x)
       else begin
                        f1 := F(t1(x));
                        f2 := F(t2(x));
                        if bal2(f1, f2) then F:= g(f1, f2)
                                        else F:= h(f1, f2)
        end
 end
```

Beraz, algoritmo errekurtsiboak badira askotan problemak ebazteko modu argi eta sinpleak, ez ordea eraginkorrak. Argitasuna ala eraginkortasuna, maiz bien artean aukera egin beharra izaten da. Zeri eman lehentasuna?

Ebazpen errekurtsiboa erraz bururatzen zaigunean, ebazpen hori diseinuaren abiapuntutzat har daiteke, hartara ebazpen iteratibo eraginkorragoa lortzeko. Azken algoritmoa ez da izango hain argia, baina ederki dokumentatua geratuko da ebazpen errekurtsiboa eta eraldakuntzan jarraitutako metodoa eransten bazaizkio.

Errekurtsibo-iteratibo programen bikurketan metodo bat baino gehiago erabil daiteke. Metodo automatikoak batzuk, eskuzkoak beste batzuk. Azken hauek problema bakoitzaren ezaugarriak era berezituan lantzeko aukera ematen dute, eta hori, ondo aprobetxatuz gero, mesedagarria izan daiteke.

Ariketak

6.1. Zenbaki bat perfektua den ala ez erabakitzen duen algoritmo errekurtsiboa diseinatu (n zenbakia perfektua da bere zatitzaile guztien batura, n ezik, n bada).

6.2. n ordenako matrize karratua emanda, matrize hori simetrikoa den ala ez erabakitzen duen algoritmo errekurtsiboa diseinatu.

6.3. Harlauztutako kale batean harlauzak zenbakituta daude 0tik hasita eta bertan ume batzuk jolasean ari dira. Umeak kalearen mutur batetik bestera doaz bi eratako mugimenduak eginez: harlauza batetik bestera mugituz edo harlauza baten gainetik salto eginez. Jokoaren hasieran umeak 0 harlauzan daude. N. harlauzara iristeko zenbat bide dagoen kalkulatzen duen algoritmo errekurtsiboa diseinatu.

6.4. Ondo eratutako zenbaki erromatar bat emanda, bere balio dezimala (zenbaki oso positiboa) kalkulatzen duen algoritmo errekurtsiboa diseinatu.
Argibideak:
Zenbaki erromatar bat R={I,V,X,L,C,D,M} multzoko elementuez osatutako sekuentzia ez-huts bat da. Rko elementuei 1, 5, 10, 100, 500 eta 1000 balioak dagozkie hurrenez hurren. Edozein zenbaki erromatar onargarrik ondoko baldintzak bete beharko ditu:
- Hiru elementu berdin baino gehiago elkarren segidan ez edukitzea.
- Bi elementu baino gehiago ez egotea elkarren segidan eta goranzko ordenean. Kasu honetan, bi elementuon balioa kalkulatzeko bigarrenari lehenengoarena kentzen zaio.

6.5. A[1..n] oso positibozko array-a eta p posizioa ($1 \leq p \leq n$) emanda, zenbat ondoz-ondoko jauzi eman behar diren erabaki nahi da, p-tik abiatuz, array-aren azkeneko posizioa gainditzeko. A[i] balioak, i posiziotik egindako jauziak zer luzera edukiko duen adierazten du. Esate baterako:

p = 2 eta A= (5, 2, 4, 1, 3, 6) hartuz,

jauzien kopurua 3 da.
A eta p emanda, egin beharreko jauzien kopurua lortzen duen algoritmo errekurtsiboa diseinatu.

6.6. N eta S zenbaki oso positiboak emanda, S bateko dituzten N digitu bitarreko sekuentzia desberdinen kopurua lortzen duen algoritmo errekurtsiboa diseinatu.

6.4. BURSTALL-EN METODOA

Errekurtsibo-iteratibo bihurketa metodo hau funtzioetan erabiltzen da, ez prozeduretan. Metodoaren gakoa errekurtsioaren inbariantea mantentzea da. Diseinatuko den iterazio baliokidearen inbariantetzat errekurtsioan gertatzen den hori hartzen da. Azken batean, algoritmo errekurtsibotik eta horri atxekitutako inbariantetik iterazio baliokidea eratorri egiten da.

Eraldatuko diren funtzio errekurtsiboak bi kasutan bana daitezke:

a) Kasu Sinplea

b) Kasu Ez-elkarkorra

Metodo honetan bost pauso ematen dira. Banan-banan ikusiko ditugu ondoko abibidean:

a) KASU SINPLEA:

Bi zenbaki osoen biderkadura kalkulatzen duen algoritmoan oinarrituko gara metodoaren xehetasunak azaltzeko:

```
function BIDERKA (x, y : integer): integer;
    begin {y ≥ 0}
        if y = 0 then BIDERKA := 0
        else if not (odd(y)) then BIDERKA := BIDERKA (2*x, y/2)
        else BIDERKA := BIDERKA (2*x, (y-1)/2) + x
end {BIDERKA = x*y}
```

1) Jeneralizazioa: Urrats honetan errekurtsioaren propietatea (asertzio induktiboa, azken batean) ezartzen da, gerora iterazioaren inbariantea izango dena. Horretarako aldagai berriak erabiltzen dira eta aldagai sartu berri hauek aldagai lokalak izango dira funtzio iteratiboan.

Errekurtsioaren propietatea asmatzeko egokia izaten da jatorrizko funtzioaren konputazioren bat aztertzea:

```
BIDERKA (25, 10)  =  BIDERKA (50, 5)
                  = BIDERKA (100, 2) + 50
                  = BIDERKA (200, 1) + 50
                  = BIDERKA (400, 0) + 200 + 50
                  = BIDERKA (400, 0) + 250
                  = 250
```

Zein izango da dei desberbinek kontserbatzen duten asertzio induktiboa? Bada, era honetakoa: BIDERKA (x, y) = BIDERKA (?, ?) + ?

Asertzio hori adierazteko aldagai-identifikadore berriak erabiliko ditugu:

Inbariantea: BIDERKA (x, y) = BIDERKA (a, b) + d

Sortuko dugun funtzio iteratiboaren inbariantea izango da asmatu dugun hori. Beraz, hemendik aurrera iterazioa eratortzen saiatuko gara, inbariante hori oinarri dela:

```
begin
        hasieraketa;
        while  ¬ test  do {BIDERKA (x, y) = BIDERKA (a, b) + d}
                  S ;
        emaitza
end
```

2) Testa: Batetik, iterazioaren irteera-baldintza finkatzen da, funtzio errekurtsiboko kasu nabariarekin edo kasu nabariekin (bat baino gehiago izan daitezke) egokitzen dena. Bestetik, itzuli beharreko emaitzak, edo kasukako emaitzak, erabakitzen dira.

Darabilgun adibidean:

test = (b = 0), hortaz: b = 0 → BIDERKA (a, b) = 0

Inbariantean: BIDERKA (x, y) = BIDERKA (a, b) + d = 0 + d = d

Horrek esan nahi du aurreko eskeman jarri dugun emaitza ondoko ekintza bihurtuko dela:

```
emaitza = (BIDERKA := d)
```

Orain artekoak bilduz:

```
function BIDERKA (x, y : integer): integer;
var a, b, d: integer;
        begin
                hasieraketa;
                while b <> 0 do {BIDERKA (x, y) = BIDERKA (a, b) + d}
                        S;
                BIDERKA := d
        end
```

Ondorengo bi pausoetan iterazioaren gorputza, S, eratortzen da:

3) Destolesketa: Inbariantea konserbatzen dela jakinda, aldagaiak dei errekurtsibo batetik bestera nola aldatzen diren ikusten da, azken batean, aldaketa horiexek bait dira iterazio-pauso batetik bestera gertatuko direnak. Funtzioaren ordezpen funtzionala egiten da inbariantean, kasu nabariak kontuan hartu gabe:

```
BIDERKA (x, y)  = [   if not (odd(b)) then BIDERKA (2*a, b/2)
                      else BIDERKA (2*a, (b-1)/2) + a ] + d
                =     if not (odd(b)) then BIDERKA (2*a, b/2) + d
                      else BIDERKA (2*a, (b-1)/2) + a + d
```

4) Bateraketa: Destolesketan lortutako adierazpena eta inbariantea bateratzean datza. Bateraketa honetarako inbarianteko aldagaiek balio eguneratuak dauzkate (iterazio-pauso baten ondorengoak). Beharrezkoa da balio eguneratudun aldagaiak eta besteak bereiztea.

Errepara dieaziogun gure adibideari:

BIDERKA (x, y) = BIDERKA (2*a, b/2) + d baldin not (odd(b))
BIDERKA (2*a, (b-1)/2) + a+ d baldin odd(b)
= BIDERKA (a', b') + d'

Beraz:

$$a' = 2 * a$$

$$b' = \left\{\begin{array}{ll} b / 2 & \text{baldin not (odd(b))} \\ (b - 1) / 2 & \text{baldin odd(b)} \end{array}\right\} = b \text{ div } 2$$

$$d' = \left\{\begin{array}{ll} d & \text{baldin not (odd(b))} \\ a + d & \text{baldin odd(b)} \end{array}\right\}$$

Balio zahar eta berrituen arteko berdintzak finkatu ondoren ez da zaila S gorputzak zer nolako esleipenak bildu behar dituen finkatzea:

```
S =  begin
       if odd(b) then d:=a+d;
       b := b div 2;
       a := 2*a
     end
```

5) Hasieraketa: Aldagai berriak hasieran inbariantea kontserbatzeko moduan hasieratu behar dira.

a := x; b := y; d :=0;
(a = x ∧ b = y ∧ d = 0)→BIDERKA (x, y) = BIDERKA (a, b) + d

Lortutako funtzio iteratiboa, honenbestez:

```
function BIDERKA (x, y : integer): integer;
var a, b, d: integer;
    begin
      a := x; b := y; d :=0;
      while b <> 0 do {BIDERKA (x, y) = BIDERKA (a, b) + d}
       begin
         if odd(b) then d:=a+d;
         b := b div 2;
         a := 2*a
       end;
      BIDERKA := d
    end
```

Ikusi dugu nola erabil daitekeen Burstall-en metodoa kasu sinple partikular batean. Gatozen orain egindako arrazonamenduei orokortasuna ematera:

```
function F(x̄ : <mota>) : <emaitzaren-mota>;
begin
    if A(x̄) then F := NAB(x̄)
    else F := F(T(x̄)) • d(x̄)
end
```

non • eragiketa bitarra den.

Funtzio-eskema hori emanda, Burstall-en bihurketa-metodoa honela gauzatuko litzateke:

1.- Jeneralizazioa:

Propietate inbariantea ezartzen da: $F(\bar{x}) = F(\bar{u}) \bullet \bar{z}$

$\bar{u}$ eta $\bar{z}$ funtzio iteratiboan erabiliko diren aldagai lokalak dira, funtzio errekurtsiboan existitzen ez zirenak.

2.- Testa:

Bukaera-baldintza eta itzuli beharreko emaitza finkatzen dira.

Bukaera-baldintza → $A(\bar{u})$

Emaitza → $F(\bar{x}) = F(\bar{u}) \bullet \bar{z} = NAB(\bar{u}) \bullet \bar{z}$

Bi lehenengo urratsak emanda funtzio iteratiboaren eskemak itxura hau hartzen du:

```
begin
    hasieraketak (ū, z̄, x̄);
    while not (A(ū)) do  {F(x̄) = F(ū)•z̄ }
        S;
    F := NAB(ū)•z̄
end
```

3.- Destolesketa:

Deiaren ordez gorputza jartzea inbariantean:

$$F(\bar{x}) = F(\bar{u}) \bullet \bar{z} = (F(T(\bar{u})) \bullet d(\bar{u})) \bullet \bar{z}$$

4.- Bateraketa:

Aldagaien balioek iterazio-pausoero izango duten gaurkotzea agerian uztea:

$$\begin{aligned} F(\bar{x}) &= (F(T(\bar{u})) \bullet d(\bar{u})) \bullet \bar{z} \\ &= (F(T(\bar{u})) \bullet (d(\bar{u}) \bullet \bar{z}) \\ &\qquad \text{(berdintza hau • eragiketa elkarkorra denean gertatuko da)} \\ &= F(\bar{u}') \bullet \bar{z}' \end{aligned}$$

Hortaz:

$$\overline{u}' = T(\overline{u})$$
$$\overline{z}' = d(\overline{u})\bullet\overline{z}$$

S iterazio-gorputzean jarriko diren aginduek $\overline{u}$-n $\overline{u}'$-ren balioa utzi beharko dute eta $\overline{z}$-ri $\overline{z}'$ esleitu beharko diote.

5.- Hasieraketa:

Iteraziora lehenengo aldiz sartzean inbarianteak bete egin behar du, beraz bistan da $\overline{u}$ eta $\overline{z}$ hasieratu egin behar direla propietate hori gerta dadin:

$$\overline{u} := \overline{x}; \quad \overline{z} := \varepsilon,$$

non ε, $\bullet$ eragiketaren elementu neutroa izango den:

$$F(\overline{x}) = F(\overline{u})\bullet\overline{z} = F(\overline{x})\bullet\varepsilon = F(\overline{x})$$

Lortutako algoritmoa:

```
begin
    u := x;   z := ε;
    while not (A(u)) do  {F(x) = F(u)•z}
            begin
                z := d(u)•z;
                u := T(u)
            end;
    F :=NAB(u)•z
end
```

Kasu nabari bat baino gehiago balego bukaera-baldintza disjuntzioa izango litzateke eta emaitza kasuka definitu beharko litzateke. Kasu orokorrak ere izan daitezke bat baino gehiago, eta horrela gertatuz gero inbarianteak kasu guztiak bildu behar ditu, destolesketan kasukako banaketa egin behar da eta bateraketa bereizitako kasuen arabera garatu behar da.

Ikus dezagun orain zer gertatzen den eragiketa elkarkorra ez denean:

b) KASU EZ-ELKARKORRA:

Ondoko funtzio errekurtsiboak, n eta b emanda, non $n \geq 0$ eta $1<b<10$ diren, n-ren adierazpidea lortzen du b oinarrian:

```
function      F(n, b): integer): integer;
    begin     {n ≥ 0 ∧ 1 < b < 10}
        if    n = 0 then F:= 0
        else  F:= 10*F(n div b, b) + n mod b
    end
```

Hasteko, kasu sinplean bezalaxe aplikatuko dugu Burstall-en metodoa:

1.- Jeneralizazioa:

$$F(n,b)=10*F(u,z)+t$$

2.- Testa:

$$u=0 \rightarrow F(n,b)=10*0+t=t$$

3.- Destolesketa:

$$\begin{aligned} F(n,b) &= 10*F(u,z)+t \\ &= 10*(10*F(u \underline{\text{div}}\ z,z)+u \underline{\text{mod}}\ z)+t \\ &= 10*F(u',z')+t' \end{aligned}$$

4.- Bateraketa:

$$\begin{aligned} F(n,b) &= 100*F(u \underline{\text{div}}\ z,z)+(10*(u \underline{\text{mod}}\ z))+t \\ &= 10*F(u',z')+t' \end{aligned}$$

Baina, kontuz!, kasu honetan ezin liteke bateraketarik burutu. Zergatik ote?

Hara, kasu honetan:

$$F(T(n,b))\bullet d(n,b) = 10*F(T(n,b))+n \underline{\text{mod}}\ b$$

hau da, • eragiketa eredu honetakoa da:

$$X\bullet Y=10*X+Y$$

Bistan da eragiketa hori ez dena elkarkorra:

$$(X\bullet Y)\bullet Z=(100*X+10*Y+Z)\neq$$
$$X\bullet(Y\bullet Z)=(10*X+10*Y+Z)$$

Horrelako arazoekin topo egindakoan irtenbidea inbariante orokorragoa asmatzea izaten da, hartara • eragiketa bateragarria bihur dadin. Gatozen adibidera:

1.- Jeneralizazioa:

$$F(n,b)=k*F(u,b)+t$$

(ikusi nola b ez den aldatzen)

2.- Testa:

$$u=0 \rightarrow F(n,b)=t$$

3.- Destolesketa:

$$\begin{aligned} F(n,b) &= k*F(u,b)+t= \\ & \quad k*(10*F(u \underline{\text{div}}\ b,b)+u \underline{\text{mod}}\ b)+t \end{aligned}$$

4.- Bateraketa:

$$\begin{aligned} F(n,b) &= k*10*F(u \underline{\text{div}}\ b,b)+k*(u \underline{\text{mod}}\ b)+t= \\ & \quad k'*F(u',b)+t' \end{aligned}$$

Hortaz:

k'=k*10
u'=u div b
t'=k*(u mod b)+t

Berdintza horiek gauzatu daitezen ondoko aginduak erabiliko ditugu:

```
t:=k*(u mod b)+t;
k:=10*k;
u:=u div b;
```

5.- Hasieraketa:

```
k:=1;  u:=n;  t:=0.
```

Pauso guztiak emanda honela geratu da funtzio eraldatua:

```
function    F(n,b:integer):integer;
    var     k,u,t:integer;
    begin   {n ≥ 0 ∧ 1 < b < 10}
        k:=1;  u:=n;  t:=0;
        while   u ≠ 0  do    {F(n,b) = k*F(u,b) + t}
            begin
                t:=k*(u mod b)+t;
                k:=10*k;
                u:=u div b
            end;
        F:=t
end
```

Ariketak

Ondoko funtzio errekurtsiboak iteratibo bihurtu Burstall-en metodoa erabiliz:

6.7.

```
function POL (A:array [0..n] of integer; x,i:integer):integer;
  begin  {0≤i≤n}
       if  i=n   then POL:=A[n]
                 else POL:=A[i] + POL(A,x,i+1)*x
  end;  {POL=A[i]+A[i+1]*x+...+A[n]*x^(n-i)}
```

6.8.

```
function BATKAR (n,m:integer):integer;
  begin {n,m≥0}
       if  m<n    then BATKAR:=0
                  else if m=n then BATKAR:= sqr(n)
                                 else BATKAR:= sqr(n)+2*BATKAR(n,m-1)
  end;  {BATKAR = Σ_{i=n}^{m} 2^i}
```

6.9.

```
function FIB (n:integer):integer;
  begin {n≥0}
       if n=0 v n=1 then FIB :=n
                    else FIB:= FIB(n-1)+FIB(n-2)
  end; {FIB= fib_n}
```

6.10.

```
function zatidura (x,y:integer):integer;
  begin {x≥0 ∧ y>0}
       if  x<y    then zatidura:=0
                  else if x=y then zatidura:=1
                                  else if not(odd(x)) and not(odd(y))
                                          then zatidura:=zatidura(x/2,y/2)
                                          else zatidura:=zatidura(x-y,y)+1
  end; {∃r(0≤r<y ∧ x=y*zatidura+r)}
```

6.11. HANOI prozedurak k disko mugitzeko zenbat mugimendu egin beharko lituzkeen kalkulatzen duen funtzio errekurtsiboa diseinatu eta ondoren iteratibo bihurtu.

6.12. Ondoko funtzio errekurtsiboak A[1..n] array ordenatuan x elementuaren bilaketa burutzen du, elementuaren posizioa itzuliz array-an badago eta 0 ez badago:

```
function bilatu (A:array [1..n] of integer; x,i,j:integer):integer;
    begin
      if i>j then bilatu:=0
             else if x=A[(i+j) div 2] then bilatu:=(i+j) div 2
                  else if  x<A[(i+j) div 2] then bilatu:=bilatu(A,x,i,(i+j) div 2-1)
                    else bilatu:=bilatu(A,x,(i+j) div 2+1,j)
end;
```

BIBLIOGRAFIA

Alagic, S., Arbib, M.A.
"The design of well-structured and correct programs".
Springer-Verlag, 1978.

Apt., K.R.
"Ten Years of Hoare's Logic: A Survey--Part I".
A. C. M. Transactions on Prog. Lang. and System, Bol 3, 4. zb.,
Urria 1981, pp 431-483.

Arsac, J.
"Foundations of Programming".
Academic Press, 1985.

Backhouse, R.C.
"Program Construction and Verification".
Prentice-Hall, 1986.

de Bakker, J.
"Mathematical Theory of Program Correctness".
Prentice-Hall, 1980.

Bird, R.S.
"Notes on Recursion Elimination".
Comm. of the A. C. M. 20 (6), Ekaina 1977, pp 434-439.

Botella, P., Rodriguez, H.
"Programación estructurada (Un intento de clarificación)".
Novatica 28, Uztaila-Abuztua, 1979.

Burstall, R.M., Darlington, J.
"A Transformation System for developping Recursive Programs".
Journal of the A. C. M., 24 bol, 1 zb, 1977, pp 44-67.

Dahl, O.J., Dijkstra, E.W., Hoare, C.A.R.
"Structured Programming".
Academic-Press, 1972.

Dahl, O.J., Owe, O.
"A Presentation of the Specification and Verification Project ABEL".
Research Report in Informatics, 90 zb., Aza. 1984. Institutt for Informatikk,
University of Oslo.

Dijkstra, E.W.
"Correctness concerns and, among other things, why they are resented".
Int. Conf. Reliable Software, SIGPLAN Notices 10, Eka. 1975, pp 546-550.

Dijkstra, E.W.
"Guarded Commands, Nodeterminancy and Formal Derivation of Programs".
Comm. to the A. C. M., Bol 18, 3 zb., Aug 1975.

Dijkstra, E.W.
"A Discipline of Programming".
Prentice-Hall, 1976.

Enderton, H.B.
"A Mathematical Introduction to Logic".
Academic Press, 1972.

Floyd, R.
"Assigning meaning to programs".
Proc. of Symp. on Applied Mathem., 1967, pp 19-32.

Goguen, J.A.
"More Thoughts on Specifications and Verifications".
A. C. M. SIGSOFT, Bol 6, 3 zb., 1981, pp 38-41.

Gries, D.
"The Science of Programming".
Springer-Verlag, 1981.

Hoare, C.A.R.
"Proof of a Program:FIND".
Comm. of the ACM, Bol 14, Urt. 1971, pp 39-45.

Hoare, C.A.R.
"Proof of Correctness of Data Representation".
Acta Informatica 1, 1972, pp 271-281.

Hoare, C.A.R., Wirth, N.
"An Axiomatic Definition of the Programming Language PASCAL".
Acta Informática 2, 1973, pp 335-355.

Johnston, H.
"Learning to Program".
Prentice-Hall, 1985.

Loeckx, J., Sieber, K.
"The Foundations of Program Verification".
Wiley-Teubner, 1984.

Lucas, M., Peyrin, J.P., Scholl, P.C.
"Algorítmica y representación de datos. 1.-Secuencias, Autómoatas de estados finitos".
Ed. Masson, 1985.

Orejas, F.
"Verificacion de Programas con Tipos estructurados de Datos".
Informática y Automática, 45 zb., 1980.

Scholl, P.C.
"Algoritmica y representacion de datos. 2.- "Recursividad y árboles".
Ed. Masson, 1986.

Wirth, N.
"The Programming Language PASCAL".
Acta Informática 1, 1971, pp 35-63.

Wirth, N.
"Program Development by Stepwise Refinement".
Comm. of the ACM, Bol 14, Ap 1971, pp 221-227.