WHILE PROGRAMAK

Konputagarritasun Teoria oinarritzeko tresna

Jesús Ibáñez; Arantza Irastorza; Ana Sánchez

Aurkibidea

1. Sarrera

Informatika Teorikoaren helburua sistema konputazionalen propietate unibertsalak edo informazio prozesamenduaren mekanismoak zuzentzen dituzten legeak definitzea eta frogatzea da. Eta unibertsalak esaten dugunean beren baliozkotasun ingurunea gure esperientzia zientifikoa baino zabalagoa duten propietateei buruz ari gara: bere baldintza teknologiko, ekonomiko eta kulturalekin *gaur egun ezagutzen dugun* Informatikan bakarrik aplikagarriak diren legeak ez dira interesgarriak, *etorkizunean izan daitekeen* Informatikan ere aplika daitezkeenak baizik. Horregatik, aurrerapen zientifiko eta teknologikoak direla medio, konputazioaren inguruan gure ikuspuntua zabaltzen joan arren baliozkoak izaten jarraituko duten oinarriak ezartzen saiatzen da.

Informatika Teorikoaren barnean, Konputagarritasun Teoriaren asmoa sistema konputazionalen *muga teorikoak* aztertzea da. Bere helburu nagusia problemak **konputagarri** eta **konputaezinen** artean bereiztea da, problema konputagarria *ebazpide informatikoa* onartzen duenari deitzen diogula kontuan hartuta. Helburu hau bi zentzutan har daiteke (eta hartzen da): *positiboan*, problema konputagarrien klase gero eta zabalagoak aurkituz, *noraino hel gaitezkeen* egiaztatuz, eta *negatiboan*, beste problema klase batzuen konputaezintasuna frogatzeko tekniken diseinuaz, *nondik ezin daitekeen pasa* zehaztuz.

Lehen kasuan metodoa nahiko zuzena da eta Informatika Aplikatuaren hainbat adarretan erabiltzen den berbera da: problemaren bat tresna informatikoen bidez ebatz daitekeela frogatu nahi badugu, tresna horiek zeintzuk diren zehaztu beharko dugu. Adibidez, kinielak asmatzearen problema konputagarria dela frogatu nahi badugu, hori egiaztatuko duen algoritmoa aurkitu beharko dugu. Horretarako problemaren inguruan dugun eta garrantzizkoa iruditzen zaigun ezagutza guztia erabil ahal izango dugu, posible den edozein konputagailutan egikaritu daitekeen edozein programazio lengoaia erabiltzerik izango dugu, beharrezkoak izango ditugun elementu teknologiko guztiak eta ebazpena aurkitzeko gehien komeni zaigun sistema informatikoa (zenbat eta ahaltsuagoa hobeto) erabil ahal izango ditugu. Are gehiago, hala behar izanez gero problema ebazteko konputagailu berezi bat garatzerik ere izango dugu. Laburtuz, garrantzizkoa izango den bakarra aurkitutako algoritmoak benetan kinielen emaitza guztiak asmatzen dituela frogatzea izango da.

Baina Konputagarritasun Teoriak arazoaren alde zailari ekiten dio: Informatikaren muga teorikoak aurkitzea, ebazpidea programatu ezin zaien

problemak aurkitzea. Demagun kinielak asmatzearen problema konputaezina dela frogatu nahi dugula, nola egin dezakegu? Pertsona batek, ehunek edo milak urte bat, ehun edo mila urte kinielak konputagailuz asmatzen saiatzen eta lortu gabe pasatzeak, aberastea zaila dela besterik ez du frogatzen. *Erabiltzen saiatu garen algoritmoetatik batek berak ere* problema ez duela ebazten egiaztatzea ez da nahikoa, *algoritmo posibleetatik bat ere ez* dela kinielak asmatzeko gai frogatu beharko dugu. Gainera beste arazo bat ere badugu: *konputagailu zehatz batean* posible diren programetatik batek ere arrakastarik ez duela egiaztatzea ere ez da nahikoa izango, problema hori bera ordenadore desberdin batean konputagarria izan litekeelako, agian etorkizun urrunean teknologikoki bideragarria izan litekeen ordenadore batean. Zailtasun hauei irtenbidea emateko Konputagarritasun Teoriak jarraitu duen estrategia honakoa izan da:

- *konputagailu eredu abstraktu* bat aukeratu eta estandar bezala hartzen da beste guztiak honetan islada daitezen. Historikoki gehien erabilitakoa Turing-en Makina izan da
- ditugun ereduen artean bat ere Turingen Makina baino hobea ez dela egiaztatzen da, hau da, konputagailu batekin (erreala edo abstraktua) konputagarria den oro makina horrekin ere hala dela. Turingen Makina konputazio sistema elektroniko guztiak baino lehenagokoa denez, egiaztapen hori aurrera joan ahala egin da: konputagailu berri bat diseinatu edo eraiki den bakoitzean, egin izan den lehen gauza Turingen Makinarekin alderatzea izan da
- zenbait problema klase ebazteko gai den Turingen Makinarik ez dagoela frogatzen da, batez ere algebra eta logika matematikotik eratorritako teknikak erabiliz. Hemendik ondoriozta dezakegu ezagunak diren konputagailu ereduetatik bat ere ez dela horiek ebazteko gai eta beraz problema horiek konputaezinak direla.

Estrategia hau paradigma desberdin askorekin praktikan jartzeak Informatika hiru ondorio nabarmenetara eraman du:

- beraien artean oso-oso desberdinak izan arren, orain arte diseinatutako konputagailu eredu guztiek konputagarritasunerako gaitasun berbera dute: bere garaian Turingen Makinarentzat frogatu zena
- Turingen Makina eta beste eredu baliokide batzuk hain sinpleak direla ikusirik, gaitasun osoa izateko ordenadore batek behar dituen eragiketa minimoak oso gutxi direla onartu beharra daukagu. Hortik, ohizko makina edo programazio lengoaien ezaugarrietatik gehienak baztergarriak direla

ondoriozta dezakegu, ezaugarri horiek zenbait lan hobeto, azkarrago edo era erosoago batean egiten laguntzen duten arren, ez baitute lan berriak egikaritzea lortzen

- problemak ebazteko gaitasunari dagokionez, egun ditugun konputagailuen (eta beraz Turingen Makinaren) gainetik jarriko litzatekeen ordenadoreren bat etorkizunean azaltzea erabat zaila da, gutxienez informazio digitalaren egungo kontzeptua erabiltzen jarraitzen dugun bitartean.

Murriztaile diren zenbait ezaugarri frogatzeko elementu garrantzitsua da Turingen Makinaren sinpletasuna, zeren eta konputazio sistema bat zenbat eta konplikatuago izan berarekin zenbait lan egikaritzea ezinezkoa dela frogatzea are eta zailagoa baita (nahiz eta frogapenak berdin balio izan). Baina badu desabantaila nabarmen bat: ohituta gauden tresna informatikoekin alderatuta bere programazio sistema arraroa da eta beraz praktikan erabiltzea oso zaila egiten da. Arazo hau konpontzeko eta estandar alternatibo moduan erabiltzeko asmotan beste sistema batzuk definitu izan dira. Testu honek horietako bat aurkezten du: while programak. Egia esan beren sintaxi klasikoa aldatu dugu, kontzeptuak irakasteko Pascal-aren ordez (ohizko while programen oinarria) ADA erabiltzen duen Ingeniaritza Informatikoaren curriculum batean pentsatuz.

While programekin Turingen makinekin ebazten diren problema berak ebazten dira (hau frogatzea testu honen helburuetatik at geratzen da). Aldiz, while programak erabiltzen askoz errazagoak dira, batez ere aurretik informatika errealean esperientzia duten pertsonentzat, lengoaia agintzaile klasikoen programen itxura hartzen baitute.

Ondorengo kapituluetan while programak zer diren eta nola erabiltzen diren (**2**. kapitulua) azalduko dugu. Oso sinplea den programazio lengoaia osatzen dutenez, agindu konplexuagoak (**3**. kapitulua) edo datu-mota aberatsagoak (**4**. kapitulua) zergatik ez ditugun sartu azaltzen saiatuko gara. Beharrezkoak ez direla ikusiko dugu, agindu edo datu-mota horiek lengoaiaren beste elementu batzuen bidez simulagarriak direlako.

2. While programak

Gure helburua **osoa** eta **minimoa** izango den programazio lengoaia definitzea da, *osoa* oinarrizko elementu guztiak edukitzea nahi dugulako eta *minimoa* oinarrizkoa ez den elementurik egotea nahi ez dugulako. Elementu bat ez da oinarrizkoa lengoaiatik ken badaiteke honen potentzia murriztu gabe, berak betetzen zituen funtzioak lengoaiaren gainontzeko elementuek (eraginkortasun gehiago edo gutxiagorekin) bete ditzaketelako. Horrela, elementua oinarrizkoa da algoritmoren bat adierazteko ezinbestekoa baldin bada, elementu hori gabe algoritmoa adierazterik ez badago. Adibidez, begizta egitura bakarra (**loop** erakoa) eta batura eta biderkadura eragiketa numerikoak lituzkeen programazio lengoaia ez-errekurtsibo batean azken eragiketa hori ez litzateke oinarrizkoa izango, biderkadura begizta eta batura erabiliz programatu daitekeelako. Egitura ziklikoa, aldiz, oinarrizkoa da begiztak ezin direlako gainontzeko kontrol egitura ez-iteratiboen bidez simulatu eta bestalde, egitura ziklikorik gabe adierazi ezin den algoritmo asko dago.

Gainetik begiratuta lengoaiaren osotasuna lortzea hau minimoa izatera iristea baino zailagoa dirudi. Alde batetik, ezinbestekoa den primitibaren bat ahaztea gerta litekeelako (batez ere oso kasu berezietan bakarrik beharrezkoa bada) eta bestalde, makina berrien diseinuak eta problema berrien planteamenduak gure lengoaia zaharkituta utz dezakeelako. Baina praktikan egoera hau ez da gertatzen: oinarrizko elementuen kopurua benetan oso txikia da, denak asko erabiltzen direnak, eta gainera, 30 hamarkadaren inguruan definitu zirenetik oinarrizko elementu berriak aurkitzeko ez da inor gai izan, eta ez duela inork inoiz lortuko ia ziurtzat ematen da. Horrela bada, lengoaia benetan minimoa, guztiz ezinbestekoa ez den ezer ez duena, lortzea da gure arazorik handiena.

Definituko dugun lengoaia oso sinplea da eta bere sintaxia ADArenean oinarrituta dago. Baina ADA dugun lengoaia barrokoenetako bat da eta gure lengoaia minimoarentzat interesatzen ez zaigun eta baztertu egingo dugun elementu mordoa du. Konkretuki, ADArekin (edo goi-mailako edozein lengoaiarekin) alderatuta, gure lengoaia sinplifikatzeko jarraitutako lerro nagusienak honakoak izango dira:

- datu-mota bakarra erabiliko da: aurretik finkatutako alfabeto baten gainean definitutako karaktere kateez (string-ez) osatuta

- edozein programatan erabilitako aldagai guztiak datu-mota horretakoak izango direnez, horiek erazagutzea ez da behar izango
- gainera aldagai horiek izendatzeko edozein identifikatzaile erabiltzea ez dugu baimenduko: sinplifikatzearren aldagai bakoitzak zenbaki (indize) bat izango du, zalantzarik sortu gabe identifikatuko duena
- konstante bezala, ε katea hutsa baino ez da erabiltzerik egongo. Horrela, programetan beste edozein erabili nahi izanez gero, ε katearen gainean jarraian egindako eragiketen bidez eraikitzea beharrezkoa izango da
- oinarrizko eragiketa bakarrak hauexek izango dira: katea bati ikur bat gehitzea edo kentzea (ezkerretik beti ere), katea hutsa den galdetzea eta bere lehen ikurraz galdetzea
- edukiko dugun agindu bakarra esleipena da
- gure lengoaian kontrolerako hiru egitura, hots, programa baten aginduak konbinatzeko hiru bide bakarrik onartuko dira: sekuentziamendua (zenbait agindu jarraian egikaritzea), baldintzatutako adarrak (zenbait agindu bide alternatibotik egikaritzea) eta iterazioa (zenbait agindu behin eta berriz egikaritzea)
- esleipen baten eskuineko aldean kabiatutako adierazpenik ez da onartuko; adibide batekin esanda, katea bati zortzi ikur gehitzeko zortzi esleipen agindu jarraian erabili beharko ditugu
- baldintzazko eta begizta kontrol egituretan dauden baldintzek ere kabiatutako adierazpenik ez dute onartuko; berez, begiztek katea hutsarekin konparazioa bakarrik eta baldintzazko egiturek lehen ikurraren egiaztaketa soilik erabil dezakete
- gure programek emaitza bakarra itzuliko dute (bat baino gehiago sortu nahi badugu, programa bat baino gehiago egikaritu beharko dugu)
- sarrera/irteera prozedurarik ez da egongo: alde batetik, egikaritzapena hastean datuak aldagai konkretu batzuetan kargatuta daudela eta bestetik, itzulitako emaitza egikaritzapena amaitutako unean beste aldagai berezi batean dagoela suposatuko dugu
- erabiltzailearen objektu klaserik (konstanteak, motak, azpiprogramak, ...) ezin izango da definitu. Horregatik eta aurretik aldagaien inguruan esandakoarengatik, gure programek inongo erazagupen motarik beharko ez dutela ondorioztatzen da

2.1. While programen Sintaxia

Aurretik, *n* ikur duen $\Sigma = \{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ alfabetoa definituta dugula suposatzen dugu. Orokorrean alfabetoaren zehaztu gabeko ikur bati buruz ari bagara ***s*** deituko dugu. Σ alfabetoaren gainean bere ikurrez osatutako hitz edo katea finituen multzoa definitu ohi da, Σ^* bezala adierazten duguna. Σ^* multzoan beti ε katea hutsa dugu (inongo ikurrik ez duena), eta Σ^*-koak diren *v* eta *w* edozein bi katea emanda, bien kateamendua $v \bullet w$-ren bidez adierazten dugu. Era berean, *x* kateak duen ikur kopurua aipatzeko $|x|$ erabiltzen dugu, eta x^R *x* katea alderantziz idaztearen ondorioz sortutako hitza adierazteko.

While programek datu moduan Σ^* multzoko kateak hartuko dituzte eta beraiekin lan egin ondoren emaitza bezala Σ^* multzoko beste katea bat itzuliko dute. While programen lengoaia osatuko duten elementu morfologikoak honakoak dira:

- **Aldagaiak**, beren izena X eta jarraian zenbaki arrunt bat izango duelarik (**X0**, **X1**, **X2**, **X185**, **X59848** aldagaien adibideak dira).
- katea hutsa adierazten duen **ε konstantea**.
- **cons**$_S$ eta **cdr** oinarrizko **funtzioak**, programek kateak maneiatzeko erabil ditzaketenak (***s*** ikurrak alfabetoaren edozein ikur adierazten du, horrela Σ alfabetoan dagoen ikur kopuru adina $cons_S$ funtzio dago). $cons_S$: $\Sigma^* \rightarrow \Sigma^*$ funtzioak edozein katearen ezkerreko aldean ***s*** ikurra gehitzen du eta *cdr*: $\Sigma^* \rightarrow \Sigma^*$ funtzioak, berriz, alderantzizko eragiketa egiten du, katearen lehen ikurra kentzen du, katea edozein dela ere. Formalki adieraziz:

$$\text{cons}_S(w) = \mathbf{s} \bullet w$$

$$\text{cdr}(w) = \begin{cases} \varepsilon & \text{baldin } w = \varepsilon \\ v & \text{baldin } w = \mathbf{s} \bullet v \end{cases}$$

  cdr(ε) arbitrarioki definitzen dugu errore-kasuak ekiditeko asmotan.

- **car**$_S$**?** eta **nonem?** oinarrizko **predikatuak**, programek kateen gainean baldintzak ebaluatzeko erabil ditzaketenak (**s** ikurrak alfabetoaren edozein ikur adierazten duenez, Σ alfabetoan dagoen ikur kopuru adina $car_S?$ predikatu dago). $car_S?$ egiazkoa da ***s*** ikurraz hasten diren kateentzat, eta *nonem?* hutsaren desberdinak diren kateentzat.
- **Puntuazio ikurrak** **:=** , **;** , **,** , **(** eta **)** , programa baten elementu desberdinak banatzen dituztenak.

- **Erreserbatutako hitzak**, **while**, **loop**, **if**, **then** eta **end**, programari egitura ematen diotenak.

Ez goiburukorik ez erazagupenik erabiliko ez dugunez, while programa batek egikarituko diren aginduen taldea, gorputza, izango du soilik. Agindu bakar batez osatutako programen kasua ere kontuan hartu behar denez, while programen definizio induktiboa egingo dugu, oinarrizko programak zeintzuk diren eta, hauetan oinarrituz eta kontrol egituren bidez, programa konposatuak nola eraikitzen diren adieraziz.

1. DEFINIZIOA: $\Sigma = \{a_1, \ldots, a_n\}$ alfabetoa emanda, Σ-ren gaineko **while programa** induktiboki honela definitzen dugu:

I) Baldin $i \in \mathbf{N}$ orduan **XI := ε;** while programa da.

II) Baldin $i, j \in \mathbf{N}$ eta $\mathbf{s} \in \Sigma$ orduan $\mathbf{XI := cons_s(XJ);}$ while programa da.

III) Baldin $i, j \in \mathbf{N}$ orduan **XI := cdr (XJ);** while programa da.

IV) Baldin P_1 eta P_2 while programak badira orduan $\mathbf{P_1\ P_2}$ while programa da.

V) Baldin $i \in \mathbf{N}$, $\mathbf{s} \in \Sigma$ eta Q while programa bada orduan hau ere while programa da **if** $\mathbf{car_s?(XI)}$ **then Q end if;**

VI) Baldin $i \in \mathbf{N}$ eta Q while programa bada orduan honako hau ere while programa da **while nonem?(XI) loop Q end loop;**

1. ADIBIDEA: $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ alfabetoa izanik, honakoak Σ-ren gaineko while programen adibideak dira:

a)
X0 := $\mathrm{cons_a}$(X0); X0 := $\mathrm{cons_c}$(X0); X0 := $\mathrm{cons_b}$(X0);
X0 := $\mathrm{cons_a}$(X0); X0 := $\mathrm{cons_b}$(X0);

b)
```
X0 := cons_a(X0);
while nonem?(X0) loop
        X0 := cons_c(X0);
end loop;
```

c)
```
X0 := ε;
while nonem?(X1) loop
        if car_b?(X1) then
                X0 := cons_c(X0);
        end if;
        X1 := cdr(X1);
end loop;
```

OHARRA: $\mathbf{P_1}$, $\mathbf{P_2}$ eta $\mathbf{P_3}$ while programak badira, **IV** klausula jarraian bi aldiz aplikatuz sortzen den $\mathbf{P_1\ P_2\ P_3}$ programa identifikatzerakoan arazo txiki bat dugu.

Definizioa zentzu hertsian hartuta, $\mathbf{Q} = \mathbf{P_1}\ \mathbf{P_2}$ eta $\mathbf{R} = \mathbf{P_2}\ \mathbf{P_3}$ izanik, $\mathbf{Q}\ \mathbf{P_3}$ eta $\mathbf{P_1}\ \mathbf{R}$ *ez dira programa bera*, bi programak era desberdinean sortuak baitira eta beraz *desberdinak*. Baina hurrengo atalean azalduko dugunez, bi while programek *portaera bera dute* horregatik $\mathbf{P_1}\ \mathbf{P_2}\ \mathbf{P_3}$ zehatz mehatz zein den jakiteak ez du gehiegi axola. Desberdintasunak egitea beharrezkoa izango ez den arren, hemendik aurrera, eta kontrakoa esaten ez dugun bitartean, $\mathbf{P_1}\ \mathbf{P_2}\ \mathbf{P_3}$ notazioa, $\mathbf{R} = \mathbf{P_2}\ \mathbf{P_3}$ izanik, $\mathbf{P_1}\ \mathbf{R}$ erako programari dagokiolako hitzarmena hartuko dugu. Horrela **1a** adibideko while programa $\mathbf{X0{:=}cons_a(X0);\ P}$ erakoa da non **P** bere aldetik $\mathbf{X0{:=}cons_c(X0);\ Q}$ erakoa den. Aldi berean **Q**ren itxura $\mathbf{X0{:=}cons_b(X0);\ R}$ da, **R** programa $\mathbf{X0{:=}cons_a(X0);\ X0{:=}cons_b(X0);}$ while programa konposatua izanik.

2.2. While programen Semantika

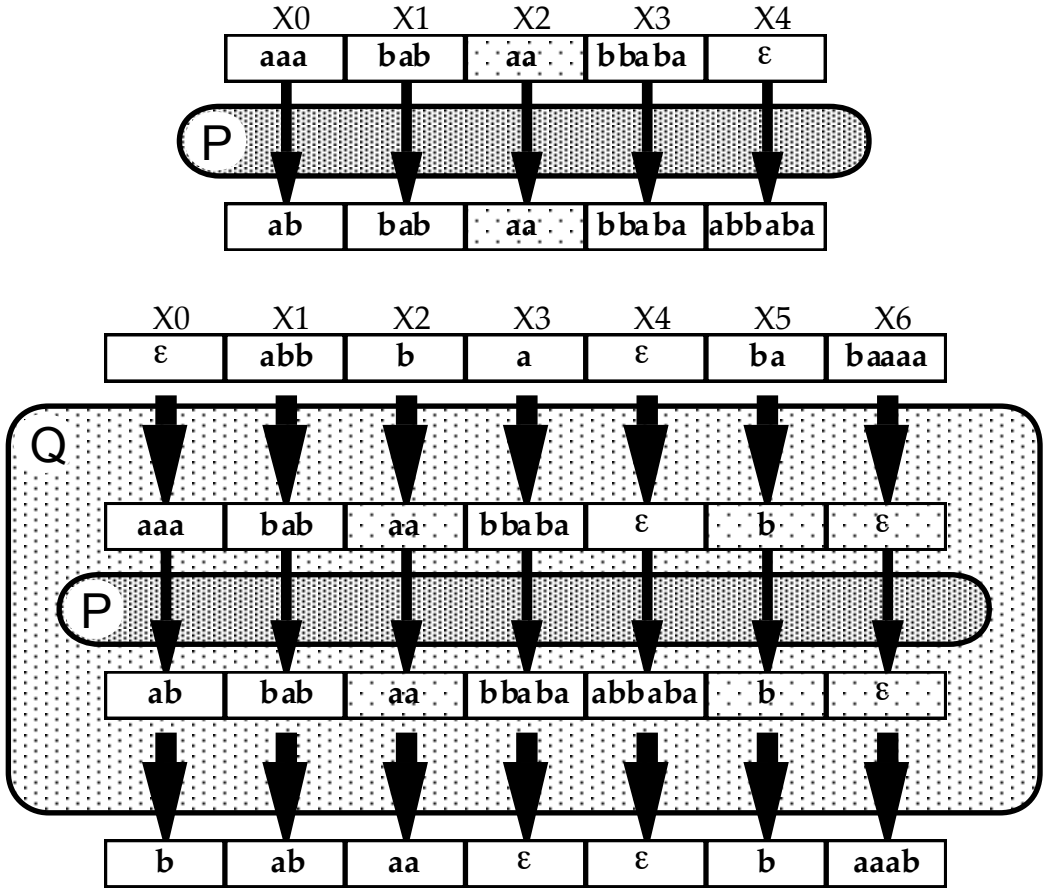
While programak eraikitzeko era definitu ondoren, nola interpretatu behar diren azaltzea beharrezkoa da (intuitiboki imajina dezakegun arren). Kasu honetan eragiketa motako semantika erabiltzea aukeratu dugu: programa, aldagai desberdinen edukinek definitzen duten ingurune konkretu batean egikaritzen dela suposatzen dugu, eta programak exekutatzen dituen eragiketak, banan bana, deskribatu behar ditugu, eta era berean ondorioz ingurunea nola aldatzen den ere adierazi behar dugu.

Horretarako lehendabizi ingurune hori deskribatzea beharrezkoa zaigu, eta helburu horrekin konputazio-egoeraren kontzeptua sartzen dugu. Gero programa baten egikaritzapenak egoera horretan duen eragina adierazi behar dugu, eta horretarako konputazioaren edo konputazio-egoeren sekuentziaren ideia sartzen dugu. Programa, bere eragiketa bakoitza exekutatzen den heinean, sekuentziaren konputazio-egoera horietatik igarotzen da. Konputazio finituak (azken agindua egikaritu ondoren bukatzen direnak) eta infinituak (begiztaren batetik atera ezin diren programen aginduen etengabeko exekuzioei dagozkienak) bereiztea garrantzizkoa da.

2. DEFINIZIOA: While programa batek ($k \geq 0$) **k+1 aldagai erabiltzen** duela esango dugu, bere testuan agertzen diren aldagaiak {X0, X1, X2, …, XK} multzoan baldin badaude.

Erabilitako aldagai kopurua, while programan agertzen diren aldagai guztiak benetan onartuko dituen segmentua hartzen uzten digun segurtasun tartea da. Honela 5 aldagai erabiltzen dituela baldin badakigu bere portaera aztertzeko 5 balioen bilakaera kontrolatzea *nahikoa* dela ere badakigu (X0, X1, X2, X3 eta X4 aldagaien edukinak). Baina honek ez du esan nahi balio horiek guztiak

kontrolatu *behar* direnik. Adibidez, **P** = **<u>if</u> car$_b$?(X1) <u>then</u> X4:=cons$_a$(X3); X0:=cdr(X1); <u>end</u> <u>if</u>**; while programak gure definizioaren arabera 5 aldagai erabiltzen ditu (baina ez dirudi X2 aldagaia ezertarako beharrezkoa denik).



1. Irudia: P while programa berari buruz ari garenean, programa hori kokatzen den testuinguruaren arabera aldagai gehiago edo gutxiago erabiltzen dituela esan dezakegu. Programa bera bakarrik hartzen bada bere testuan agertzen diren aldagaiak kontuan hartzea nahikoa da (kasu honetan 5), baina benetan Q beste while programa baten azpiprograma izango bada, azken honen beharretara egokitu beharko du. Orduan Q-ren testuan agertzen den aldagai adina erabiltzen duela esango dugu (kasu honetan 7). Hau, sobera dauden aldagaiak (X5 eta X6) P-renak ere balira bezala eta P-ren eginkizun zehatza beren edukina irakurri edo aldatzea ez dela bezala interpretatu behar dugu.

Era honetan, erabilitako aldagaien kopurua unibokoki determinatu gabe dagoela esatea ere komenigarria da. Esan bezala, aurreko parrafoan aipatutako **P** while programak 5 aldagai erabiltzen du (bere aldagaiak {X0, X1, X2, X3, X4} multzoan daudelako), baina, definizioarekin bat etorriz, adibidez 7 aldagai erabiltzen duela esatea ere definizioarekin bat dator (aldagaiak {X0, X1, X2, X3, X4, X5, X6} multzoan ere daudelako). Baina **P** while programak 4 aldagai erabiltzen duela ezin dugu esan (bere testuan {X0, X1, X2, X3} zerrendan ez dagoen X4 agertzen delako). Zergatik uzten dugu anbiguotasun hori definizioan? Zergatik ez dugu definizio zehatzago bat egiten, hau da, **P**-k zehatz-mehatz 5 aldagai

erabiltzen duela (eta ez 4 ezta 6 ere) esaten duen bat? **P** handiagoa den **Q** while programaren azpiprograma bat besterik ez izatea gerta daitekeelako eta **Q** while programa horren eragiketetan X5 eta X6 aldagaiak azaldu daitezkeelako. Hau horrela bada, **P**-k aldagai batzuen edukina alda dezakeela adierazteaz gain *gainontzekoen edukina ezin duela aldatu* azaltzea interesatuko zaigu (ikus 1. Irudia)

3. DEFINIZIOA: Σ alfabetoa bada eta P Σ-ren gainean definitutako eta *k+1* aldagai erabiltzen dituen while programa bada, P-ren **konputazio-egoera** k+1 dimentsioa duen eta Σ^* multzoaren gainean definituta dagoen edozein bektore da. Konputazio-egoerak P-ren egikaritzapenaren une konkretu batean aldagaiek duten edukina adierazten du.

4. DEFINIZIOA: Eragiketa bat aldaketa (**XI :=** ε, **XI := cons$_s$(XJ)** edo **XI := cdr (XJ)** erako esleipena) edo kontsulta (**nonem?(XI)** edo **car$_s$?(XJ)** erako baldintza) motakoa izan daiteke. P while programaren eragiketak P-ren testuan agertzen direnak dira, programak burutu ditzakeen oinarrizko ekintzak. Horregatik gure eragiketa motako semantikaren oinarrizko elementuak izango dira.

Orain while programa baten portaera denboran zehar deskribatzen utziko digun kontzeptua aurkeztuko dugu: konputazioa. **Konputazioa** eragiketen eta konputazio-egoeren sekuentzia da, eta honako propietateak ditu:

- bere lehen elementua konputazio-egoera bat da (**hasierako egoera**)
- konputazio-egoerak eta eragiketak bata bestearen ondoren txandakatu egiten dira
- sekuentziaren egoera guztiek dimentsio bera dute *k+1*
- sekuentziaren eragiketa guztiak *k+1* aldagai erabiltzen dituen P while programari dagozkio
- sekuentzia infinitua izan daiteke, baina finitua bada bere azkeneko elementua konputazio-egoera izango da, **bukaerako egoera** deitzen duguna
- eragiketen sekuentzia P-ren barruko eragiketak egikaritzen jarraitzen den ordenari dagokio, eta konputazio-egoeren sekuentziak exekuzioaren ondorioz *k+1* aldagaien edukinean gertatutako aldaketak adierazten ditu.

  Hau da, konputazioa era honetako sekuentzia finitua da

$$V_0\, A_1\, V_1\, A_2\, V_2\, A_3\, V_3\, A_4 \ldots A_n V_n$$

eta orduan **konbergentea** dela diogu, edo honelako sekuentzia infinitua

$$V_0\ A_1\ V_1\ A_2\ V_2\ A_3\ V_3\ A_4\ \ldots\ A_n V_n\ A_{n+1}\ \ldots$$

eta **dibergentea** dela diogu. Bi kasuetan V_i bakoitzak konputazio-egoera adierazten du, eta A_i bakoitzak eragiketa. Formalki deskribatuz:

5. DEFINIZIOA: Bitez P while programa eta V_0 P-ren konputazio-egoera. P-ri eta V_0-ri dagokien K **konputazioa** induktiboki honela definitzen dugu:

I) P **XI := ε;** erako programa baldin bada, orduan:

$$\mathbf{K = V_0}\ XI{:=}\ \varepsilon\ \mathbf{V_1}$$

non V_1 honakoa betetzen duen bektorea den

$$V_1[m] = \begin{cases} V_0[m] & \text{baldin } m \neq i \\ \varepsilon & \text{baldin } m = i \end{cases}$$

hau da, XI aldagaiaren edukina adierazten duen *i*-garren koordenatuan ezik beste guztietan V_0 bektorearekin bat datorrena, eta *i*-garren koordenatu horretan ε balioa daukana.

II) P **XI := cons$_s$(XJ);** erako programa baldin bada, orduan:

$$\mathbf{K = V_0}\ XI{:=}\ cons_{\mathbf{s}}(XJ)\ \mathbf{V_1}$$

non V_1 honakoa betetzen duen bektorea den

$$V_1[m] = \begin{cases} V_0[m] & \text{baldin } m \neq i \\ cons_{\mathbf{s}}(V_0[j]) & \text{baldin } m = i \end{cases}$$

aurreko kasuaren antzera.

III) P **XI := cdr(XJ);** erako programa baldin bada, orduan:

$$\mathbf{K = V_0}\ XI{:=}\ cdr(XJ)\ \mathbf{V_1}$$

non V_1 aurreko bi kasuen antzera honela definitzen den bektorea den:

$$V_1[m] = \begin{cases} V_0[m] & \text{baldin } m \neq i \\ cdr(V_0[j]) & \text{baldin } m = i \end{cases}$$

IV) P **P$_1$ P$_2$** erako programa baldin bada, orduan bi kasu dugu :

1) K_1 konputazio-egoera eta eragiketen sekuentzia infinitua izanda, $V_0\ K_1$ segida P_1-i eta V_0-ri dagokien konputazio dibergentea bada, orduan:

$$\mathbf{K = V_0\ K_1}$$

P_1 while azpiprogramak ziklatu egiten du eta P_2 ez da exekutatzen hasten, eta beraz K dibergentea da.

2) $V_0\ K_1\ V_1$ konputazioa P_1-i eta V_0-ri dagokien konputazio konbergentea bada (K_1 eragiketa eta konputazio-egoeren sekuentzia finitua izanda)

eta $V_1\ K_2$ konputazioa P_2-ri eta V_1-i dagokien konputazioa bada (K_2 eragiketa eta konputazio-egoeren sekuentzia finitua edo infinitua izanik), orduan:

$$\mathbf{K = V_0\ K_1\ V_1\ K_2}$$

P_1-ek konbergitzen du eta P_2 egikaritzen da, eta beraz K konbergentea izango da $V_1\ K_2$ ere hala denean.

V) P **<u>if</u> car$_s$?(XI) <u>then</u> Q <u>end</u> <u>if</u>;** erako programa bada bi kasu posible dugu:

1) car$_s$?(V_0[i]) egiazkoa ez bada, orduan:

$$\mathbf{K = V_0}\ \text{car}_s?(\text{XI})\ \mathbf{V_0}$$

Baldintza ebaluatu egiten da eraginik sortu gabe, V_0 egoeran faltsua izanda Q while azpiprograma exekutatu behar ez delako. Kasu honetan K konbergentea izango da.

2) car$_s$?(V_0[i]) egiazkoa bada, orduan bedi $V_0\ K_1$ konputazioa Q-ri eta V_0-ri dagokiena (non K_1 finitua edo infinitua izan daitekeen eragiketen eta konputazio-egoeren sekuentzia den). Kasu honetan:

$$\mathbf{K = V_0}\ \text{car}_s?(\text{XI})\ \mathbf{V_0\ K_1}$$

V_0 egoeran baldintza egia dela ikusi ondoren Q exekutatu egiten da. K konbergentea edo dibergentea izango da $V_0\ K_1$-en arabera.

VI) P **<u>while</u> nonem?(XI) <u>loop</u> Q <u>end</u> <u>loop</u>;** erako programa bada, V_{j+1} bektoreari Q-ri eta V_j-ri dagokien $V_j\ K_{j+1}\ V_{j+1}$ konputazio konbergentearen bukaerako konputazio-egoera deituko diogu. Dagokien konputazioa dibergentea bada, gerta daiteke V_{j+1} bektorea *j* balioren batentzat (eta beraz handiagoak direnentzat ere) ez existitzea. Hiru kasu posible bereizten dugu:

1) *j* balioren bat (0 bera ere izan daiteke) existitzen bada zeinentzat nonem?(V_j[i]) ez den betetzen, orduan bedi *m* propietate hori duen balio txikiena, hots, $V_m[i]=\varepsilon \wedge \forall j\ (\ j<m \rightarrow V_j[i]\neq\varepsilon\ )$ betetzen duena. Kasu honetan:

$$\mathbf{K = V_0}\ \text{nonem?}(\text{XI})\ \mathbf{V_0\ K_1\ V_1}\ \text{nonem?}(\text{XI})\ \mathbf{V_1\ K_2\ V_2\ \ldots}$$
$$\mathbf{\ldots\ V_m}\ \text{nonem?}(\text{XI})\ \mathbf{V_m}$$

XIren edukina ez da hitz hutsa Qren *m*-garren iterazioa bukatu arte, une horretan begiztatik ateratzen da. K konputazio konbergentea da.

2) Balio hori existitzen ez bada eta V_{j+1} egoera *j*-ren balio guztientzat definituta badago, orduan:

$$\mathbf{K = V_0}\ \text{nonem?(XI)}\ \mathbf{V_0\ K_1\ V_1}\ \text{nonem?(XI)}\ \mathbf{V_1\ K_2\ V_2}\ \ldots$$
$$\ldots\ \mathbf{V_j}\ \text{nonem?(XI)}\ \mathbf{V_j\ K_{j+1}\ V_{j+1}}\ \ldots$$

egikaritzapena ez da behin ere bukatzen Q while azpiprogramaren infinitu iterazioek irteera baldintza egiazkoa izatea ez baitute lortzen. Argi dagoenez K dibergentea da.

3) Balio hori existitzen ez bada eta *j*-ren balioren batentzat V_{j+1} indefinituta badago, orduan bedi *m* propietate hori duen balio txikiena. V_m existitzen bada eta V_{m+1} berriz ez, Q eta V_m-ri dagokien konputazioa dibergentea delako, eta beraz V_m K_{m+1} erakoa delako da (K_{m+1} sekuentzia infinitua izanda). Kasu honetan:

$$\mathbf{K = V_0}\ \text{nonem?(XI)}\ \mathbf{V_0\ K_1\ V_1}\ \text{nonem?(XI)}\ \mathbf{V_1\ K_2\ V_2}\ \ldots$$
$$\ldots\ \mathbf{V_m}\ \text{nonem?(XI)}\ \mathbf{V_m\ K_{m+1}}$$

Q while azpiprogramaren m+1-garren iterazioak ziklatzen du. Kasu honetan ere K dibergentea da.

2. ADIBIDEA: Konputazio konbergentearen eta dibergentearen adibide bana ikus dezagun. **1c** adibideko programa eta (**ca**, **bab**) hasierako egoera-bektorea hartuz:

(**ca**, **bab**) X0:=Ɛ (ε, **bab**) nonem?(X1) (ε, **bab**) $\text{car}_{\mathbf{b}}$?(X1) (ε, **bab**)
X0:=$\text{cons}_{\mathbf{c}}$(X0) (**c**, **bab**) X1:=cdr(X1) (**c**, **ab**) nonem?(X1) (**c**, **ab**)
$\text{car}_{\mathbf{b}}$?(X1) (**c**, **ab**) X1:=cdr(X1) (**c**, **b**) nonem?(X1) (**c**, **b**) $\text{car}_{\mathbf{b}}$?(X1) (**c**, **b**)
X0:=$\text{cons}_{\mathbf{c}}$(X0) (**cc**, **b**) X1:=cdr(X1) (**cc**, ε) nonem?(X1) (**cc**, ε)

eta **1b** adibidekoa (ε, **cba**) bektorearekin

(ε, **cba**) X0:=$\text{cons}_{\mathbf{a}}$(X0) (**a**, **cba**) nonem?(X0) (**a**, **cba**) X0:=$\text{cons}_{\mathbf{c}}$(X0) (**ca**, **cba**)
nonem?(X0) (**ca**, **cba**) X0:=$\text{cons}_{\mathbf{c}}$(X0) (**cca**, **cba**) nonem?(X0) (**cca**, **cba**)
X0:=$\text{cons}_{\mathbf{c}}$(X0) (**ccca**, **cba**) nonem?(X0) (**ccca**, **cba**) ...

2.3. While-konputagarritasuna

Aurreko atalean while programa baten *barneko* portaera, hau da, karaktere-kateak bere aldagaietan *prest dituenean* nola lantzen dituen azaldu dugu. Bere *kanpoko* portaera azaltzea geratzen zaigu, hau da, konputazioa hasi baino lehen, nola lortzen da hasierako konputazio-egoera osatzeko datuak sartzea?. Eta bukatu ondoren, bukaerako konputazio-egoera nola interpretatzen da lortutako emaitza zein den jakiteko?.

Lehen ere aipatu dugu gure while programek datu bat baino gehiago har dezaketela, baina emaitza bakarra itzultzen dutela, eta sarrera-irteerarako aldagai berezi batzuk edukiko ditugula. Zehazki, while programa egikaritzen hasten denean bere datuen balioak X1, ..., XN aldagaietan (behar adina) "automatikoki" kargatuta dituela, eta bukatzen duenean emaitza X0 aldagaian uzten duela suposatuko dugu. Programak aipatutakoak baino aldagai gehiago erabil dezake (lanerako aldagaiak), eta hasieraketa-erroreak ekiditeko daturik ez duten aldagai guztiak ε balioarekin hasieratuta daudela suposatuko dugu.

Horrela, *k+1* aldagai erabiltzen duen eta *n* datu hartzen dituen while programa batek, hasieran **X0 := ε; get(X1); get(X2); ...; get(XN); X(N+1):= ε; ...; XK:= ε;** aginduak eta bukaeran **put(X0)** agindua *baleuzka bezala lan egiten du*. Hitzarmen honen bidez sarrera eta irteera automatikoak direla suposatzen ari gara, eta beraz irakurketa eta idazketa prozeduren beharrik ez dugu.

Beraz, while programa baten kanpoko portaera sarrera/irteeraren bidez deskriba daiteke, datuen eta emaitzaren arteko erlazioa adieraziz. Hau sarrera posibleen multzotik irteera posibleen multzora doan funtzio baten bidez egin ohi da. Konputazioa dibergentea bada while programak ziklatu dezakeenez, funtzio hori definituta ez egotea ere gerta daiteke.

OHARRA: Funtziorako erabiliko dugun kontzeptua **funtzio partziala** izango da. Honekin adierazi nahi dugu funtzioa datu posible guztien gainean definituta egon daitekeela edo ez. *j* argumentu duen ψ funtzio partziala $\psi: \Sigma^{*j} \longrightarrow \Sigma^*$ idazten dugu. ψ funtzioa $(y_1,y_2,\ldots,y_j)$ *j*-tuplaren gainean definituta badago (emaitza bat sortzen badu) **konbergitzen** duela esaten dugu eta $\psi(y_1,y_2,\ldots,y_j)\downarrow$ idazten dugu. Definituta ez badago (emaitzarik sortzen ez badu) **dibergitzen** duela esaten dugu eta $\psi(y_1,y_2,\ldots,y_j)\uparrow$ idazten dugu. Funtzio partziala **osoa** (argumentu posible guztientzat konbergitzen badu) edo **ez-osoa** (argumenturen batentzat dibergitzen badu) izan daiteke.

ψ funtzioa konbergitzen duen balioen multzoa *dom(ψ)* eremua da, eta funtzioak sor ditzakeen balioen multzoa *ran(ψ)* heina da. Funtzioak izendatzeko φ, ψ, χ, etab. letra grekoak erabiliko ditugu, baina funtzioak *osoak direla dakigunean* eta hori azpimarratu nahi dugunean f, g, h, etab. letra latinoak hartuko ditugu.

P while programa batek *k+1* aldagai erabiltzen badu, gure arazoa da har ditzakeen datuen kopurua, *n*, zein den nola jakin erabakitzea. Berriro ere irtenbiderik sinpleena hartuko dugu: *n* kopurua ez du **P** programak aurretik ezartzen, eta horrela programa, emandako datuen kopuruaren arabera era desberdinetan erabil daiteke. Hala, *programa bat benetan bat baino gehiago da*, bere

erabilpen desberdin bakoitzeko bat: while programa baten portaera ez da berdina izango datu bakarra duenean, edo zazpi dituenean, edo sarrerarik ez duenean. While programa bakoitza funtzioen ebaluatzaile gisa interpreta daiteke, argumentu kopuru posible bakoitzeko funtzio desberdin bat burutuz.

6. DEFINIZIOA: Demagun Σ alfabetoaren gainean *k+1* aldagai erabiltzen duen **P** while programa definituta dugula, eta bedi *j≥0*. Datuen eta emaitzaren arteko erlazioa deskribatzen duen $\varphi_P^j : \Sigma^{*j} \longrightarrow \Sigma^*$ funtzio partzialari **P-k konputaturiko funtzio j-tarra** deitzen diogu. Aipatu erlazio hori *j* sarrerekin erabilitako **P** programaren konputazioek ezartzen dute, eta edozein $(y_1,y_2,\ldots,y_j)$ *j* argumenturen gainean honela definitzen da:

a1) $k \geq j$ bada, hasierako konputazio-egoera bezala $V = (\varepsilon,y_1,y_2,\ldots,y_j, \varepsilon,\ldots,\varepsilon)$ bektorea hartuko dugu

a2) $k \leq j$ bada, hasierako konputazio-egoera bezala $V = (\varepsilon,y_1,y_2,\ldots,y_k)$ bektorea hartuko dugu, eta y_{k+1}, y_{k+2}, …, y_j datuez ahaztuko gara (ezin izango dira **P**-n erabili bertan X(K+1), X(K+2), ..., XJ aldagaiak agertzen ez direlako)

b1) **P** eta V-ri dagokien konputazioa konbergentea bada, eta bukaerako konputazio-egoera $V' = (z_0,z_1,\ldots,z_k)$ bada, orduan, $\varphi_P^j(y_1,y_2,\ldots,y_j) = z_0$ (funtzioa balio horientzat definituta dago)

b2) **P** eta V-ri dagokien konputazioa dibergentea bada, orduan $\varphi_P^j(y_1,\ldots,y_j)\uparrow$ (funtzioa balio horientzat ez dago definituta)

3. ADIBIDEA: Bitez $\Sigma = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$ alfabetoa eta honako P while programa:

```
X0 := cons_b(X1);
X0 := cons_a(X0);
if car_c?(X2) then
      X0 := cdr(X2);
      while nonem?(X3) loop
            X0 := cons_a(X4);
            if car_c?(X4) then
                  X3 := cdr(X3);
            end if;
      end loop;
end if;
```

Beste edozein while programak bezala P-k infinitu funtzio konputatzen du. Lehendabiziko seiak (*j* 0 eta 5en artean) ondoren definitzen ditugu:

$\varphi_P^0 = \mathbf{ab}$

$$\varphi_P^1(u) = \mathbf{ab}\bullet u$$

$$\varphi_P^2(u,v) = \begin{cases} \mathrm{cdr}(v) & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \\ \mathbf{ab}\bullet u & \text{bestela} \end{cases}$$

$$\varphi_P^3(u,v,x) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{baldin } \neg\mathrm{car}_{\mathbf{c}}?(v) \\ \mathrm{cdr}(v) & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \wedge x = \varepsilon \\ \bot & \text{bestela} \end{cases}$$

$$\varphi_P^4(u,v,x,y) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{baldin } \neg\mathrm{car}_{\mathbf{c}}?(v) \\ \mathrm{cdr}(v) & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \wedge x = \varepsilon \\ \mathbf{a}\bullet y & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \wedge x \neq \varepsilon \wedge \mathrm{car}_{\mathbf{c}}?(y) \\ \bot & \text{bestela} \end{cases}$$

$$\varphi_P^5(u,v,x,y,z) \simeq \begin{cases} \mathbf{ab}\bullet u & \text{baldin } \neg\mathrm{car}_{\mathbf{c}}?(v) \\ \mathrm{cdr}(v) & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \wedge x = \varepsilon \\ \mathbf{a}\bullet y & \text{baldin } \mathrm{car}_{\mathbf{c}}?(v) \wedge x \neq \varepsilon \wedge \mathrm{car}_{\mathbf{c}}?(y) \\ \bot & \text{bestela} \end{cases}$$

Sarrerako argumentu kopuru handiagoko funtzio guztiek emaitza bera itzuliko dute (sobera daudenak ez dituzte inoiz kontuan hartuko).

OHARRA: $\bot$ ikurrarekin emaitza hutsa adieraziko dugu, funtzioak ez duela emaitzarik itzultzen. Hots, $\psi(x)\uparrow$ eta $\psi(x) \simeq \bot$, bi adierazpen hauekin gauza bera esaten dugu (baina $\psi(x) \simeq \uparrow$ jartzea ez dago ondo, $\uparrow$ eta $\downarrow$ ez baitira balioak, postfixu bidezko notazioan adierazitako predikatuak baizik). Bestalde, **berdintasun partziala**ren ikurra $\simeq$ erabiltzen dugu indefinituta egon daitezkeen adierazpenak maneiatzen ditugulako. Ildo honetan, $\psi(x) = \chi(y)$ baieztatu izan bagenu *aldi berean hiru gauza* esango genukeen:

$$\psi(x) = \chi(y) \Leftrightarrow \begin{cases} \psi(x)\downarrow \\ \wedge \\ \chi(y)\downarrow \\ \wedge \\ \psi(x) \text{ eta } \chi(y) \text{ hitz bera dira} \end{cases}$$

Horregatik $\psi(x) = \perp$ adierazpena *beti faltsua* izango da, $\psi(x)$ konbergentea izan ala ez. Indefinituak izan daitezkeen balioak konparatu ahal izateko berdintasun partziala behar dugu, hau konparatutako bi adierazpenak indefinituak direnean ere egiazkoa delarik:

$$\psi(x) \simeq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\downarrow \wedge \chi(y)\downarrow \wedge \psi(x) \text{ eta } \chi(y) \text{ hitz bera dira} \\ \vee \\ \psi(x)\uparrow \wedge \chi(y)\uparrow \end{cases}$$

Berdintasunek funtzioak beraien artean oso-ososrik konparatzeko ere balio dute, lehen aipatutako printzipioak errespetatuz beti ere: berdintasuna konbergentzian betetzen da. Beraz:

$$\psi = \chi \Leftrightarrow \forall x\, \psi(x) = \chi(x)$$

$$\psi \simeq \chi \Leftrightarrow \forall x\, \psi(x) \simeq \chi(x)$$

Bigarren kasuan bi funtzioak bai konbergitzen dutenean bai dibergitzen dutenean berdinak direla esaten ari gara eta beraz, eremu bera dutela eta eremu horretako elementuen gainean emaitza berak sortzen dituztela. Baina lehenengoan biek *beti konbergitzen* dutela (osoak direla) eta gainera balio guztietan bat datozela esaten ari gara. Hortaz, ψ funtzioa ez-osoa bada $\psi = \psi$ adierazpena *ez da egiazkoa izango* baina $\psi \simeq \psi$ beteko da, ψ-ren izaeraren menpe egon gabe.

Azpimarratzekoa da $=$ berdintasunarentzat esan duguna defini ditzakegun predikatu guztientzat ere orokorrean egiazkoa dela, eta bereziki desberdintasunarentzat $\neq$. Hau egiazkoa izango da alderatutako bi adierazpenak *definituta badaude eta desberdinak diren hitzei badagozkie:*

$$\psi(x) \neq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\downarrow \\ \wedge \\ \chi(y)\downarrow \\ \wedge \\ \psi(x) \text{ eta } \chi(y) \text{ hitz desberdinak dira} \end{cases}$$

Hortaz, indefinituta dauden, edo bat konbergentea eta bestea dibergentea diren, bi adierazpen alderatzen ditugunean berdinak ez direla, ezta desberdinak ere, ikusten dugu ($\perp = \perp$ eta $\perp =$ **aba** adierazpenak ez dira egiazkoak baina beraien "aurkakoak" ere ez, $\perp \neq \perp$ eta $\perp \neq$ **aba**). Horregatik *desberdintasuna ez da berdintasunaren ukapena*. Haatik **desberdintasun partziala** erabil dezakegu:

$$\psi(x) \neq \chi(y) \Leftrightarrow \begin{cases} \psi(x)\downarrow \wedge \chi(y)\downarrow \wedge \psi(x) \text{ eta } \chi(y) \text{ hitz desberdinak dira} \\ \vee \\ \psi(x)\downarrow \wedge \chi(y)\uparrow \\ \vee \\ \psi(x)\uparrow \wedge \chi(y)\downarrow \end{cases}$$

eta hau *bai*, berdintasun partzialaren ukapena da ($\psi(x) \neq \chi(y) \Leftrightarrow \neg\psi(x) \simeq \chi(y)$).

Funtzioak oso-osorik konparatzeko azken desberdintasun hori bakarrik erabiltzen dugu, baita berdintasun partzialaren ukapena bezala ere.

$$\psi \neq \chi \Leftrightarrow \exists x\, (\psi(x) \neq \chi(x))$$

7. DEFINIZIOA: $\psi: \Sigma^{*j} \rightarrow \Sigma^*$ funtzio partziala **while-konputagarria** dela diogu bera konputatzen duen P while programa, hau da, $\varphi_P^j \simeq \psi$ betetzen duena, existitzen bada.

4. ADIBIDEA: Honako funtzioak while-konputagarriak dira:

a) cdr eta $cons_s$ (edozein $s \in \Sigma$) funtzioak while-konputagarriak dira. Hori frogatzen duten programak, hurrenez hurren, **X0:=cdr(X1);** eta **X0:=cons$_s$(X1);** dira.

b) Funtzio hutsa $\perp\!\!\!\perp(x_1,x_2,...,x_j) \simeq \perp$, beti indefinituta dagoena. Bere while-konputagarritasuna frogatzen duen programetako bat **1b** adibidekoa da.

c) $K_w(x_1,x_2,...,x_j) = w$ erako edozein funtzio konstante, argumentu kopurua, *j*, edozein izanda ere. K_ε konputatzen duen programa **X0:=ε;** izan daiteke eta **P** programak K_x konputatzen badu orduan **P X0:=cons$_s$(X0);** programak $K_{s \bullet x}$ konputatuko du. **1a** adibideko while programak $K_{\mathbf{babca}}$ funtzio konstantea konputatzen du, argumentu kopurua edozein izanda.

d) Identitate funtzioa, $id:\Sigma^* \rightarrow \Sigma^*$, id(x) = x, while programa honek konputatzen du (***s*** ikurra Σ alfabetoko edozein ikur izanda):

```
X0 := cons_s(X1);
X0 := cdr(X0);
```

e) $p_k^j : \Sigma^{*j} \rightarrow \Sigma^*$, $p_k^j(z_1, \ldots, z_j) = z_k$, non $1 \leq k \leq j$, argumentu konkretu bat bereizten duten proiekzio-funtzioak, antzeko eran konputatzen dira:

```
X0 := cons_s(XK);
X0 := cdr(X0);
```

f) *lehena*: $\Sigma^* \rightarrow \Sigma^*$ funtzioa, ikur bakarreko hitza itzultzen duena ikur hori bere sarrera-argumentuaren lehendabizikoa izanik (hitz hutsaren kasuan ε itzultzea erabaki dugu):

X0 := ε;
if $car_{\mathbf{a_1}}$(X1) **then** X0:= $cons_{\mathbf{a_1}}$(X0); **end if**;
…
if $car_{\mathbf{a_n}}$(X1) **then** X0:= $cons_{\mathbf{a_n}}$(X0); **end if**;

Konputagarritasun Teoriaren barnean garrantzi berezia duen eta aparteko tratamendua jasotzen duen funtzio azpitalde bat dugu. Funtzio horiek *kalkulu* bati (edozein emaitza sor dezake baita emaitzarik eza ere) lotuta egon ordez, *erabaki* batekin erlazionatuta daude (emaitza beti sortzen dute, baina honek BAI/EZ erako erantzun batekin erlazionatu ohi diren bi balio posible bakarrik ditu). Egia esan funtzio boolearrak edo *predikatuak* dira, datu batzuen gainean EGIAZKOA balioa eta gainontzekoen gainean FALTSUA itzultzen dutenak. While programek balio boolearrik itzuli ezin dutenez, datu-mota hori erabiltzen ez dutelako, funtzio horiek maneiatu ahal izateko ihesbide bat hartu behar dugu.

8. DEFINIZIOA: Bedi R *j* argumentutako predikatua. Ondoren definitzen dugun $C_R:\Sigma^{*j} \rightarrow \Sigma^*$ funtzio osoari **R predikatuaren ezaugarri funtzioa** deitzen diogu:

$$C_R(z_1,z_2,...,z_j) = \begin{cases} \mathbf{a_1} & \text{baldin } R(z_1, z_2, ..., z_j) \\ \varepsilon & \text{bestela} \end{cases}$$

non $\mathbf{a_1}$ hitza Σ alfabetoaren lehen ikurraz osatutako hitza den. C_R funtzioak R predikatua bere argumentuen gainean noiz betetzen den eta noiz ez detektatzeko balio du.

Era berean, Σ^*-eko hitz-tuplez osatutako edozein multzori predikatu bat egoki diezaiokegu, bere argumentua multzo horretakoa den kasuan bakarrik beteko den predikatu bat. Hortaz, multzoak predikatuak bezala trata ditzakegu. Benetan ez dago multzoaren eta predikatuaren kontzeptuak bereizteko pisuzko arrazoirik eta beraz bata zein bestea erabiliko dugu.

9. DEFINIZIOA: Bedi $A \subseteq \Sigma^{*j}$. Ondoren definitzen dugun $C_A:\Sigma^{*j} \rightarrow \Sigma^*$ funtzio osoari **A multzoaren ezaugarri funtzioa** deitzen diogu.

$$C_A(z_1,z_2,...,z_j) = \begin{cases} \mathbf{a_1} & \text{baldin } (z_1,z_2,...,z_j) \in A \\ \varepsilon & \text{bestela} \end{cases}$$

10. DEFINIZIOA: P multzo edo predikatu j-tarra **while-errekurtsiboa** da bere ezaugarri funtzioa $C_P: \Sigma^{*j} \rightarrow \Sigma^*$ while-konputagarria bada.

5. ADIBIDEA: Honako predikatuak while-errekurtsiboak dira (bere definizioa multzoen notazioa erabiliz idazten dugu):

- $car_{\mathbf{s}}?(x) \Leftrightarrow x \in \{ \mathbf{s}\bullet y: y \in \Sigma^* \}$

  X0 := ε; **if** $car_{\mathbf{s}}?$(X1) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**;

- nonem?(x) $\Leftrightarrow x \in \{ z: z \neq \varepsilon \}$

  X0 := ε; **while** nonem?(X1) **loop** X0 := $cons_{\mathbf{a_1}}$(X0); X1 := ε; **end loop**;

- berdin_hasi?(x,y) $\Leftrightarrow (x,y) \in \{ (x,y): \exists \mathbf{s} \in \Sigma\ (car_{\mathbf{s}}?(x) \wedge car_{\mathbf{s}}?(y)) \}$

  X0 := ε;
  if $car_{\mathbf{a_1}}$?(X1) **then if** $car_{\mathbf{a_1}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;
  if $car_{\mathbf{a_2}}$?(X1) **then if** $car_{\mathbf{a_2}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;
  …
  if $car_{\mathbf{a_n}}$?(X1) **then if** $car_{\mathbf{a_n}}$?(X2) **then** X0 := $cons_{\mathbf{a_1}}$(X0); **end if**; **end if**;

3. Makroak

Eraiki behar ditugun programen konplexutasuna gehitzen doan heinean gure programazio lengoaiaren sinpletasunak eragozpen garrantzitsua sortzen digu: dena hasiera-hasieratik programatu behar izatea. Adibidez, $f(x) = x^R$ funtzioa while-konputagarria dela frogatzeko **P** while programa idatzi dugula suposa dezagun. **P**-k ez dauka konplikatuegia izateko beharrik: **if** agindu batzuk erabili behar ditugu *x*-ren ikurrak, banan-banan, zeintzuk diren jakiteko, eta emaitzari gehitzen joateko. Orain demagun hitzak kateatzen dituen $g(x,y) = x \bullet y$ funtzioa konputagarria dela frogatu nahi dugula, eta horretarako **Q** while programa idazten dugula. *cons* funtzioak hitzak atzetik aurrera eraikitzera behartzen gaituela kontuan hartuz, *y*-ren ikurrak emaitza izango den aldagaira pasatuz hasi beharko dugu, gero *x*-ren ikurrak banan-banan gehituko dizkiogu, azkenekotik hasirik eta lehenengoarekin bukatuz. Hori lortzeko lehendabizi *x* alderanztu behar dugu, beraz **Q** programak azpiprograma gisa **P** edukiko duela suposa dezakegu. Era berean, hitzen kateaketa bitarteko eragiketa moduan behar duen edozein while programak barnean **Q**-ren agindu guztiak eduki beharko ditu eta horrela hurrenez hurren.

Gureak bezain oinarrizko eragiketen bidez algoritmoak adierazi behar izateak programen luzera nabarmenki gehitzea dakar, eta ondorioz programatu behar dituen pertsona gogaitzea. Jakina denez programazio lengoaia konbentzionaletan arazo hau kodearen modularizazio mekanismoen bidez konpondu izan da, horien artean programategi eta azpierrutinen definizioa.

Hala ere ez dirudi gure programazio lengoaiak bere sintaxian elementu berriren beharra duenik (zenbait gauza programatzea astuna izan daitekeenaz kexatzen ari garela kontuan izan eta ez, egin ezin direnaz), orain arte ikusitakoaren arabera, behintzat. Eta beharrezkoa ez bada, ziur ez zaigula interesatzen, gure lengoaiak ez bailuke minimoa izaten jarraituko. Bi interes horiek nola uztar ditzakegu?, alde batetik, berriz idazten ibili gabe, jadanik diseinatutako programak berrerabili nahi izatea, eta bestalde, guztiz beharrezkoa ez bada, while programen lengoaia zabaldu nahi ez izatea?.

Irtenbidea laburdurak erabiltzetik dator: "eta hemen joan zen astean idatzi genuen P programaren kodea letorke" erako mekanismoak. Honekin ez gara while programen sintaxia zailtzen ari (laburdurak *ez baitira* while programak), era erosoago batean deskribatzen utziko diguten hitzarmenak erabiltzen baizik. Laburdurak while programen ordezkariak dira, nahiz eta programa horiek

aurrean ez izan, *beharrezkoa izanez gero nola lortu* jakingo genuke. "Benetakoa" den gauza bakarra, aurreko kapituluan deskribatutako sintaxia erabiliz idatzitako while programa dela kontuan izan behar dugu beti. Horregatik ez dugu inoiz programa laburtu bat sartuko, atzetik while programa bat dagoela ziur ez badakigu.

Makroa while programak laburtzeko mekanismoa da. Makroaren **hedapena** makro hori erabiliz laburtutako while programa da. **Makroprograma** makroak erabiliz idatzitako programa da. Makroak definituz programa laburragoak idatz daitezke, kodearen errepikapena ekiditen delako: programa berriak idazten goazen heinean programategi antzeko batera sartzen ditugu eta hurrengo programek programategi hori erabiltzerik izango dute. Makro bat definitu ahal izateko beharrezkoa da:

- makro hori erabiltzeko beharra arrazoitzea
- bere sintaxia deskribatzea (laburdura nola eraiki eta non erabili)
- hedapen mekanismoa garatzea (makroaren bidez laburtu den while programa nola lor litekeen)

Definitzeko garaian makroaren hedapena deskribatzea garrantzizkoa da, atzetik benetako while programa duelako laburdura lasai erabil dezakegula ziurtatzen digun mekanismoa baita. Hemendik aurrera **programa** hitza while programa nahiz makroprograma esateko erabiliko dugu. Gure programetan erabiliko ditugun lau eratako makroak definitzera goaz: makroaldagaiak (**3.1** atala), makroadierazpenak (**3.2** atala), makrobaldintzak (**3.3** atala) eta kontrolerako makroegiturak (**3.4** atala). Azkenean makroak ahal den hobekien erabiltzeko zenbait hitzarmen emango ditugu (**3.5** atala).

3.1. Makroaldagaiak

Nahiz eta, zehatz mehatz, laburdura mekanismo bat ez izan, erabilitako aldagai kopurua kontrolatu behar ez izateko eta, programazio lengoaia desberdinetan ohitura denez, beren edukina adierazten duten aldagai-izenak erabiltzeko, XI erakoak ez diren aldagaiak dituzten programak idatzi nahiko ditugu noizean behin.

11. DEFINIZIOA: Makroaldagaia ADAren irizpideak jarraitzen dituen edozein identifikatzaile da (letra bat eta ondoren letra, digitu eta azpimarraren ikurrez osatutako segida ez-hutsa, azpimarraren ikur horiek ezin direlarik jarraian egon ezta bukaeran ere). Makroprograma batean makroaldagaiak ohizko

aldagaiak egoten diren posizio berberetan ager daitezke, hots, U eta V makroaldagaiak badira, makroprogrametan honako hauek bezalako aginduak ager daitezke:

- U := ε;
- U := $cons_s$(V);
- U := cdr(V);
- **if** car_s?(U) **then** P **end if**;
- **while** nonem?(U) **loop** P **end loop**;

non P-k makroaldagai gehiago izan ditzakeen.

Makroaldagaiak sarreran eta irteeran erabiltzea ekidingo dugu (hurrenez hurren, X1, X2, ..., XJ eta X0 aldagaiak erabiltzen jarraituko dugu, gaizki-ulertzeak ez izatearren) eta era berean gainontzeko kasuetan horiek erabiltzen saiatuko gara, horrela programa bakoitzean garbi egongo da zeintzuk diren lanerako soilik diren aldagaiak.

Makroaldagaiak dituen programari dagokion hedapena makroaldagai bakoitza "legezko" izena duen aldagai berri batekin ordezkatuz lortzen da. Izan bedi **P** *n+1* aldagai arrunt (X0, X1, ... , XN) eta *m* makroaldagai (V_1, V_2, ..., V_m) erabiltzen duen makroprograma. **P**-ren hedapena, **Q**, lortzeko V_i makroaldagai bakoitzaren agerpen guztiak X(N+I) aldagai berriarekin ordezkatzen ditugu. Emaitza while programa bat izango da.

6. ADIBIDEA: $f(x) = x^R$ funtzioa konputagarria dela frogatuko dugu, eta horretarako aldagai laguntzaile bat (LAG makroaldagaia) izango duen **P** makroprograma erabiliko dugu. Aldagai horretan sarrera kopiatzen dugu, ez galtzearren. Programa horretan hiru programazio ohitura aurkeztuko ditugu, beharrezkoak ez izan arren programen irakurketa argitzen baitute. Lehena sarrerako aldagaien edukina ez aldatzea izango da. Bigarrena, sarrerako aldagaiak ez direnak beti hasieratzea, sistemak defektuz ε balioa ematen diela kontuan hartu gabe. Hirugarrena, komenigarria den kasuetan begizten inbarianteak iruzkin moduan sartzea (egia esan, hirugarren honek lehenengoa eskatzen digu: sarrerako aldagaiak ez badira aldatzen, hasierako datuak inbarianteetan aipatzeko erabiltzeko aukera izango dugu).

```
    LAG := cons_a1(X1); LAG := cdr (LAG); X0 := ε;
- - LAG^R•X0=X1^R
    while nonem?(LAG) loop
        if car_a1?(LAG) then X0 := cons_a1(X0); end if;
```

if car_{a_2}?(LAG) **then** X0 := $cons_{a_2}$(X0); **end if**;
...
if car_{a_n}?(LAG) **then** X0 := $cons_{a_n}$(X0); **end if**;
LAG := cdr (LAG);
end loop;

P-ren hedapena aurkitu nahi izango bagenu, 1+1 aldagai arrunt erabiltzen duela, eta beraz ordezkapena egiteko X2 aldagaia erabil daitekeela, kontuan hartu beharko genuke. Hedapena honakoa litzateke:

X2 := $cons_{a_1}$(X1); **X2** := cdr (**X2**); X0 := ε;
while nonem?(**X2**) **loop**
if car_{a_1}?(**X2**) **then** X0 := $cons_{a_1}$(X0); **end if**;
if car_{a_2}?(**X2**) **then** X0 := $cons_{a_2}$(X0); **end if**;
...
if car_{a_n}?(**X2**) **then** X0 := $cons_{a_n}$(X0); **end if**;
X2 := cdr (**X2**);
end loop;

Jakina, definitutako makroprograma bakoitza hedatzea ez da beharrezkoa: horrela balitz ez genuke ezer aurreratuko. Orain egin dugunaren antzera, makroa definitzeko garaian bere hedapena egiteko prozesua orokorrean deskribatzen dugu eta honekin kasu konkretu bakoitzarekin prozesu hau erabil daitekeenaz konbentzituko gara. Honela, une horretatik aurrera makroa lasaitasun osoz erabiltzerik izango dugu, makroprogramaren atzean, deskribatutako hedapen prozesuaren bidez lor daitekeen while programa baliokide bat dagoela jakingo baitugu. Kapitulu honetako adibide batzuen hedapenak ematen baditugu argigarriagoa izan dadin egiten dugu, emandako hedapen mekanismoa hobeto uler dadin.

3.2. Makroadierazpenak

Behin funtzio bat konputatzen duen programa izanda, bitarteko kalkuluak egiteko funtzio horren emaitza behar duen bakoitzean programa hori bera errepikatu behar ez izatea komeni zaio programatzaileari. Irtenbidea **while**-konputagarriak direla badakigun funtzioak azpierrutinak bezala erabiltzea da, behar ditugunean dei ditzakegunak. Beste programazio lengoaietan erabiltzaileak definitutako funtzioei deiak egiten dizkienean jarraitzen den ideia bera da.

12. DEFINIZIOA: Bedi $\psi: \Sigma^{*j} \dashrightarrow \Sigma^*$ while-konputagarria den funtzioa. **Makroadierazpena** $\psi(V_1, ..., V_j)$ erako adierazpena da, eta bere argumentuak $V_1, ..., V_j$ aldagaien edukinak direnean ψ funtzioak itzultzen duen balioa bezala

interpretatzen da. V_1, ..., V_j aldagai arruntak edo makroaldagaiak izan daitezke. Makroprograma batek makroadierazpenak esleipenetan eduki ditzake, bestela esanda, esleipen arruntez gain esleipenok ere izan ditzake:

- $U := \psi(V_1, ..., V_j)$;

Makroprogramaren hedapenak esleipen hauetako bakoitza dagokion funtzio konputagarria kalkulatzen duen while programa batekin ordezkatu beharko du. Aldagaien erabilerarekin arretaz ibili behar dugu: ezin dugu ψ kalkulatzen duen programa zuzenean txertatu, makroprogramaren beste tokiren batean beharrezkoak diren datuak aldatu genitzakeelako.

Bedi **P** makroadierazpenak dituen makroprograma. **P**-ren hedapena bi pausotan egiten da. Lehendabizi $\mathbf{U:=\psi(V_1, ..., V_j);}$ erako esleipen bakoitza makroprograma honengatik ordezkatzen dugu.

$Z1 := cons_{\mathbf{s}}(V_1)$; $Z1 := cdr(Z1)$;
…
$ZJ := cons_{\mathbf{s}}(V_j)$; $ZJ := cdr(ZJ)$;
Q_Z
$U := cons_{\mathbf{s}}(Z0)$; $U := cdr(U)$;

non **Q** programa ψ funtzioa konputatzen duen *while programa* baita (*n+1* aldagai erabiltzen duela suposatuko dugu), Z0, ..., ZJ aldagaiak **P**-n agertu ez eta hedapenaren aurreko pausoetan azaldu ez diren makroaldagaiak baitira eta $\mathbf{Q_Z}$ programa ***Q**-n XI aldagai bakoitza ZI makroaldagaiarekin ordezkatu ondoren sortzen* baita (horrela bere funtzionamenduak ez du makroadierazpena dagoen makroprogramaren portaera zapuztuko). Bukatzeko, ZI erako makroaldagaiak, eta **P** programak aurretik izan zitzakeen guztiak ere, hedatu beharko dira.

Kontuan izan hedapenean lehendabizi, aldagai lokalen papera hartzen duten makroaldagaietan datuen "karga" egiten dela, gero programa bera egikaritzen dela eta bukatzeko, emaitza nahi den aldagaian "deskargatzen" dela. Horrela egin behar da **Q** programak bere datuak ez dituelako derrigorrez X1, ..., XJ aldagaietatik hartu behar, V_1, ..., V_j aldagaietatik baizik, eta batez ere, **P**-ren beste aldagaien edukina ezin delako aldatu.

7. ADIBIDEA: Honako programak f(x,y) = x•y funtzioa konputatzen du, identitatea eta alderantzizkoa funtzio konputagarriei dagozkien makroadierazpenak dituzten bi esleipen erabiliz, **X0:=X2;** eta $\mathbf{LAG:=X1^R;}$.

X0 := X2;
LAG := $X1^R$;
- - LAG^R•X0=X1•X2
while nonem?(LAG) **loop**

if car_{a_1}?(LAG) **then** X0 := $cons_{a_1}$(X0) **end if**;
if car_{a_2}?(LAG) **then** X0 := $cons_{a_2}$(X0) **end if**;
…
if car_{a_n}?(LAG) **then** X0 := $cons_{a_n}$(X0) **end if**;
LAG := cdr (LAG);
end loop;

OHARRA: Programetan identitatearen izena (id) ez aipatzeko hitzarmena hartzen dugu, aurkakoak ulermena nahastea besterik ez bailuke egingo.

Makroadierazpenen hedapenaren lehen pausoaren ondoren honako programa hau edukiko genuke

```
-- identitatea kalkulatzeko X2tik argumentuaren karga
   Z1 := cons_a1(X2); Z1 := cdr(Z1);
-- identitatearen kalkulua
   Z0 := cons_a1(Z1); Z0 := cdr(Z0);
-- identitatearen kalkuluaren emaitza X0n deskargatu
   X0 := cons_a1(Z0); X0 := cdr(X0);
-- alderantzizkoa kalkultzeko X1etik argumentuaren karga
   Y1 := cons_a1(X1); Y1 := cdr(Y1);
-- alderantzizkoaren kalkulua
   Y2 := cons_a1(Y1); Y2 := cdr (Y2);
   while nonem?(Y2) loop
        if car_a1?(Y2) then Y0 := cons_a1(Y0); end if;
        ...
        if car_an?(Y2) then Y0 := cons_an(Y0); end if;
        Y2 := cdr (Y2);
   end loop;
-- alderantzizkoaren kalkuluaren emaitza LAGen deskargatu
   LAG := cons_a1(Y0); LAG := cdr(LAG);
-- programaren gainontzekoa
   while nonem?(LAG) loop
        if car_a1?(LAG) then X0 := cons_a1(X0); end if;
        ...
        if car_an?(LAG) then X0 := cons_an(X0); end if;
        LAG := cdr (LAG);
   end loop;
```

Erabili diren programak, identitatea eta alderantzizkoa konputatzen dutenak, **4d)** eta **6)** adibideetatik hartu dira (bigarrenaren kasuan lehenik bere makroaldagaia hedatu behar izan dugu).

OHARRA: Kateamendua funtzioa while-konputagarria dela frogatu dugun unetik beste makroadierazpen batzuetan erabiltzerik izango da. Erabiltzen

dugunean, hau da, makroprograma baten testuan agertzen denean, ADAren ohiko notazioa erabiliko dugu (**&** ikurra).

1. PROPOSIZIOA: (while-konputagarriak diren funtzioen **konposaketa** while-konputagarria da). $\psi_1,...,\psi_n$: $\Sigma^{*j} \longrightarrow \Sigma^*$ *n* funtzioak eta θ: $\Sigma^{*n} \longrightarrow \Sigma^*$ funtzioa while-konputagarriak badira, orduan ondoren definitzen dugun χ: $\Sigma^{*j} \longrightarrow \Sigma^*$ konposaketa funtzioa ere while-konputagarria da:

$\chi(y_1,...,y_j) = \theta(\psi_1(y_1,...,y_j), \ ... \ ,\psi_n(y_1,...,y_j))$.

Frogapena

Honako makroprogramak χ funtzioa konputatuko luke

Z1 := ψ_1(X1, ..., XJ);
...
ZN := ψ_n(X1, ..., XJ);
X0 := θ(Z1, ..., ZN);

eta makroaldagai eta makroadierazpenen hedapena eginez while programa baliokidea lortuko genuke.

8. ADIBIDEA: $h(x,y) = x^R \bullet y \bullet \mathbf{bbb}$ funtzioa while-konputagarria da honako funtzio while-konputagarrien konposaketa bezala idatz daitekeelako.

$h(x,y) = \theta(\psi_1(x,y), \psi_2(x,y),\psi_3(x,y))$, non

$\theta(u,v,w) = u \bullet v \bullet w$

$\psi_1(x,y) = x^R$

$\psi_2(x,y) = y = p_1^2(x,y)$ (while-konputagarria)

$\psi_3(x,y) = \mathbf{bbb} = K_{\mathbf{bbb}}(x,y)$ (while-konputagarria)

ψ_2 eta ψ_3 funtzioen while-konputagarritasuna frogatuta dago, baina besterik iruditu dakigukeen arren θ eta ψ_1 funtzioena ez da frogatu. Adibidez, *bi* hitzen kateaketa while-konputagarria dela frogatu da, baina ez *hiru* hitzena hala denik. Frogatzea ez da zaila, baina horretarako proiekzio funtzioak kontuz erabili beharko dira:

$\theta(x, y, z) = x \bullet y \bullet z = \lambda(\xi_1(x,y,z), \xi_2(x,y,z))$, non

$\lambda(u,v) = u \bullet v$ (while-konputagarria)

$\xi_1(x,y,z) = x \bullet y$

$\xi_2(x,y,z) = z = p_3^3(x,y,z)$ (while-konputagarria)

Azkenik, ξ_1 ere while-konputagarria dela ikusteko beste konposaketa baten bidez lor dezakegu:

$\xi_1(x,y,z) = x \bullet y = \mu(\rho_1(x,y,z), \rho_2(x,y,z))$, non

$\mu(u,v) = u \bullet v$ (while-konputagarria)

$\rho_1(x,y,z) = x = p_1^3(x,y,z)$ (while-konputagarria)

$\rho_2(x,y,z) = y = p_2^3(x,y,z)$ (while-konputagarria)

ψ_1 konputagarria dela frogatzea ere erreza da, alderantzizkoaren eta bi osagaien lehen proiekzioaren (p_1^2) arteko konposaketa baita.

OHARRA: Makroadierazpenak eta makroaldagaiak erabiltzeak nahasketak sor ditzake. **X2:=abb;** erako makroadierazpen bat printzipioz bi eratara uler daiteke: abb izeneko makroaldagai bat badagoela eta bere edukina X2 aldagaira pasa nahi dugula edo X2n **abb** hitza gorde nahi dugula ($K_{\mathbf{abb}} = \mathbf{abb}$ funtzioa konputagarria denez makroa zuzena izango litzateke). Gaizki-ulertuak ekiditeko konstanteak beti komatxoen artean idatziko ditugu (**X2:='abb';**) eta makroaldagaientzat letra larriak bakarrik (**X2:=ABB;**) erabiliko ditugu.

3.3. Makrobaldintzak

Programa zailagoak diseinatzerakoan kontrolerako egitura iteratiboetan eta baldintzazkoetan car_s? eta nonem? predikatuak bakarrik erabil ahal izatea nahiko deserosoa gerta daiteke. Aldagaien balioak kalkulatzeko adierazpen konplexuen kalkulua laburtzea baimendu dugun bezala, gure programen kontrola gidatuko luketen baldintza konplexuak ebaluatzeko gauza bera egin ahal izatea arrazoizkoa litzateke.

Ideia berez antzekoa da. While-errekurtsiboa den edozein R predikaturi while-konputagarria den funtzio bat dagokio (bere ezaugarri funtzioa, C_R), lanerako aldagai batean kalkulatua izan daitekeena. Behin hau eginda, emaitza while programen ohizko predikatu batek (car_s? edo nonem?) azter dezake eta praktikan baldintza ebaluatua izan balitz bezala da.

13. DEFINIZIOA: Bedi $R \subseteq \Sigma^{*j}$ while-errekurtsiboa den predikatu bat. **Makrobaldintza** $R(V_1, ..., V_j)$ erako adierazpena da, bere argumentuak $V_1, ..., V_j$ aldagaien edukinak direnean R-k sortzen duen balio boolearra bezala interpretatzen dena. $V_1, ..., V_j$ aldagai arruntak edo makroaldagaiak izan daitezke. Makroprograma batek makrobaldintzak baldintzazko aginduetan edo agindu

iteratiboetan eduki ditzake, hau da, agindu arruntez gain mota honetako aginduak izan ditzake:

- **if** $R(V_1, ..., V_j)$ **then** Q **end if**;
- **while** $R(V_1, ..., V_j)$ **loop** Q **end loop**;

non **Q** beste programa bat den, bere makroaldagaiak, makroadierazpenak edota makrobaldintzak eduki ditzakeena.

Makrobaldintzak makroadierazpen bat erabiliz hedatuko dira. Horretan R predikatuari dagokion ezaugarri funtzioa makroaldagai berri batean kalkulatuko da. Bedi **P** makrobaldintzak dauzkan makroprograma bat. **P**-ren hedapena bi pausotan egiten da. Lehenik **<u>if</u> R(V_1, ..., V_j) <u>then</u> Q <u>end</u> <u>if</u>;** erako agindu bakoitza era honetako makroprograma batengatik ordezkatzen da:

```
Z := C_R(V_1, ..., V_j);
if car_a1?(Z) then Q end if;
```

<u>while</u> R(V_1, ..., V_j) <u>loop</u> Q <u>end</u> <u>loop</u>; erako agindu bakoitza berriz, era honetako makroprograma batengatik ordezkatzen da:

```
Z := C_R(V_1, ..., V_j);
while nonem?(Z) loop
      Q
      Z := C_R(V_1, ..., V_j);
end loop;
```

non Z **P**-n erabili ez den makroaldagai berria den (baina makrobaldintza guztietan Z bera erabil daiteke, kabiatuta egon arren ere: kasu bakoitzean predikatua Z-n ebaluatzen da, juxtu bere emaitza baldintza batean erabili aurretik).

Azkenik, horrela sartutako makroadierazpenak eta makroaldagaia eta **P** programak berak eduki litzakeenak hedatu behar dira.

9. ADIBIDEA: Ikusita dugunez, *berdin_hasi?*$\subseteq \Sigma^{*2}$ eta *nonem?* $\subseteq \Sigma^{*}$ predikatuak while-errekurtsiboak dira, hortaz makrobaldintzetan ager daitezke. Horrela egin dugu *aurrizkia?*(x,y) $\Leftrightarrow \exists w \ (y = x \bullet w)$ eran definitutako *aurrizkia?* $\subseteq \Sigma^{*2}$ predikatuaren ezaugarri funtzioa konputatzen duen makroprograma honetan:

```
   X0 := 'a1'; LAG1 := X1; LAG2 := X2;
-- ∃Z(Z•LAG1=X1 ∧ Z•LAG2=X2)
   while berdin_hasi?(LAG1, LAG2) loop
         LAG1 := cdr(LAG1);
         LAG2 := cdr(LAG2);
```

```
end loop;
if nonem?(LAG1) then X0 := ε; end if;
```

Lehen makrobaldintzaren hedapenaren lehen pausoaren ondoren lortutako programa honakoa litzateke:

```
X0 := 'a1';
LAG1 := X1;
LAG2 := X2;
-- baldintzaren ebaluazioa
Z := Cberdin_hasi?(LAG1, LAG2);
while nonem?(Z) loop
        LAG1 := cdr(LAG1);
        LAG2 := cdr(LAG2);
-- baldintza berriz ebaluatu
        Z := Cberdin_hasi?(LAG1, LAG2);
end loop;
if nonem?(LAG1) then X0 := ε; end if;
```

Hedapenak makroadierazpenekin eta gero makroaldagaiekin jarraituko luke. *nonem?(LAG1)* makrobaldintzaren hedapena berdin egingo litzateke.

Makrobaldintzetan predikatu errekurtsiboen erabilpena definitu ondoren, konposaketa eta eragiketa logikoak erabiliz sortzen diren adierazpen boolear konplexuagoak definitzen utziko diguten mekanismoak aurkez ditzakegu. Horretarako, while-errekurtsiboak diren predikatuen gainean mekanismo horiek aplikatuz predikatu while-errekurtsiboak sortzen direla egiaztatzea nahikoa izango da.

Konposaketari dagokionez, **1.** proposizioan azaldutako funtzioen konposaketan erabiltzen zen oinarri bera aplikatzen da, baina funtzioen ordez predikatuak hartuz. Desberdintasuna bi detailetan datza. Alde batetik, predikatu batek edozein eratako datuak har ditzake, beraz beste mota bateko funtzioekin konposatu daiteke eta ez predikatuekin derrigorrez. Baina aldi berean predikatua funtzio osoa da, hortaz konposaketaren emaitza ere hala dela ziurtatzeko konposaketan erabilitako beste funtzio horiek ere osoak izatea eskatu behar dugu.

2. PROPOSIZIOA: (Predikatu errekurtsibo baten eta *osoak* eta konputagarriak diren funtzioen arteko **konposaketa** predikatu errekurtsiboa da) $f_1,...,f_n: \Sigma^{*j} \rightarrow \Sigma^*$ funtzioak osoak eta while-konputagarriak badira, eta $R \subseteq \Sigma^{*n}$ predikatua while-errekurtsiboa bada, orduan ondoren definitzen dugun $P \subseteq \Sigma^{*j}$ predikatua ere while-errekurtsiboa da:

$$P(z_1, \dots, z_j) \Leftrightarrow R(f_1(z_1, \dots, z_j), \dots, f_n(z_1, \dots, z_j))$$

Frogapena

$C_P(z_1, \ldots, z_j) = C_R(f_1(z_1, \ldots, z_j), \ldots, f_n(z_1, \ldots, z_j))$ konputagarria dela ikusteko C_R, f_1, …, f_n funtzioei **1.** proposizioaren emaitza aplikatzea nahikoa da. Funtzio hauek guztiak osoak direnez C_P, beraien konposaketa, ere osoa da. Eta azkenik, C_R ezaugarri funtzio bat denez C_P ere hala da.

10. ADIBIDEA: atzizkia?(x, y) $\Leftrightarrow$ $\exists$w (y = w • x) eran definitutako *atzizkia?*$\subseteq \Sigma^{*2}$ predikatua errekurtsiboa da. Izan ere, atzizkia?(x, y) $\Leftrightarrow$ aurrizkia?(x^R, y^R) baita, hortaz proposizioaren baldintzak betetzen dira errekurtsiboa den *aurrizkia?* predikatua while-konputagarriak diren $f_1(x, y) = x^R$ eta $f_2(x, y) = y^R$ funtzioekin konposatzen ari garelako. Funtzio horien while-konputagarritasuna proiekzioa eta alderantzizkoa funtzio while-konputagarrien konposaketaz frogatzen da.

3. PROPOSIZIOA: Bitez $R, S \subseteq \Sigma^{*j}$ predikatu while-errekurtsiboak. Orduan bien **konjuntzio** eta **disjuntzio** predikatuak, eta bietako baten **ezeztapen** predikatua ere, horrela dira. Bestela esanda, honako predikatuak while-errekurtsiboak dira:

$T_1(y_1,\ldots,y_j) \Leftrightarrow R(y_1,\ldots,y_j) \wedge S(y_1,\ldots,y_j)$

$T_2(y_1,\ldots,y_j) \Leftrightarrow R(y_1,\ldots,y_j) \vee S(y_1,\ldots,y_j)$

$T_3(y_1,\ldots,y_j) \Leftrightarrow \neg R(y_1,\ldots,y_j)$

Frogapena

Konjuntzio predikatuaren (T_1) ezaugarri funtzioa konputatuko lukeen makroprograma honakoa litzateke:

X0 := ε;
if nonem?(C_R(X1, ..., XJ)) **then** X0 := C_S(X1, ..., XJ); **end if**;

Disjuntzio predikatuaren (T_2) ezaugarri funtzioari dagokiona:

X0 := ε;
if nonem?(C_R(X1, ..., XJ) & C_S(X1, ..., XJ)) **then** X0 := $\text{cons}_{\mathbf{a_1}}$(X0) **end if**;

Eta azkenik, ezeztapenerako (T_3):

X0 := ε; X0 := $\text{cons}_{\mathbf{a_1}}$(X0);
if nonem?(C_R(X1, ..., XJ)) **then** X0 := ε; **end if**;

OHARRA: Eragile logikoak ADAren terminologiarekin erabiliko ditugu: **and** konjuntziorako, **or** disjuntziorako eta **not** negaziorako.

11. ADIBIDEA: Berdintasun predikatua $= \subseteq \Sigma^{*2}$ while-errekurtsiboa da, errekurtsiboak diren bi predikatuen konjuntzioa bezala idatz daitekeelako:

$$=(x,y) \Leftrightarrow \text{aurrizkia?}(x, y) \wedge \text{aurrizkia?}(y, x)$$

=(x, y) jarri ordez, ohizkoagoa den eran, x = y idatziko dugu. Era berean $\neq \subseteq \Sigma^{*2}$ desberdintasun predikatuaren while-errekurtsibotasuna ondoriozta dezakegu:

$$\neq(x,y) \Leftrightarrow \neg\, x = y$$

hau ere infixu bidezko notazioan idatziko dugu, eta programaren batean agertzen bada ADAn ohizkoa den idazkera erabiliz (/= ikurra).

3.4. Kontrolerako makroegiturak

Makroadierazpenak eta makrobaldintzak definitu ondoren programatzearen lana askoz errazagoa dela egiazta dezakegu. Hala ere, goi-mailako programazio lengoaiekin, ADArekin esaterako, lan egitera ohituta dagoen jendeak ziur aski baliagarriak diren zenbait agindu faltan botako ditu. Gure hurrengo pausoa horrelako aginduak baimentzea izango da, baina makroak edo laburdurak bezala berriz ere: agindu hauek edukiko dituzten programak ez dira benetako while programak izango, eta laburtzen duten while programa zehatz-mehatz definitu beharko dugu.

3.4.1. if_then_else makroa

Faltan botatzen den lehen agindua, ziur aski, baldintza orokorrarena izango da: **if B_1 then Q_1 elsif B_2 then Q_2 ... elsif B_n then Q_n else Q_{n+1} end if;**, non $\mathbf{B_1}$, $\mathbf{B_2}$, ..., $\mathbf{B_n}$ makrobaldintzak eta $\mathbf{Q_1}$, $\mathbf{Q_2}$, ..., $\mathbf{Q_{n+1}}$ makroprogramak diren. Bere semantika ohikoa litzateke: $\mathbf{B_1}$ egiazkoa bada, $\mathbf{Q_1}$ egikaritzen da; bestela, $\mathbf{B_2}$ egiazkoa bada $\mathbf{Q_2}$ egikaritzen da eta horrela baldintza guztiekin $\mathbf{Q_n}$ arte; azkenik baldintza guztiak faltsuak badira $\mathbf{Q_{n+1}}$ programa egikaritzen da. Makro horren hedapena egiteko Z makroaldagaia sartzen da baldintzen ebaluaketa desberdinak kontrolatzeko.

Bedi **P** if_then_else motako kontrolerako makroegiturak dituen programa. **P**-ren hedapena bi pausotan egiten da. Lehenik, **if B_1 then Q_1 elsif B_2 then Q_2 ... elsif B_n then Q_n else Q_{n+1} end if;** erako agindu bakoitza era honetako makroprograma batekin ordezkatzen dugu:

Z := ε; Z := cons_s(Z);
if B_1 **then** Q_1 Z := ε; **end if**;

if B_2 **and** nonem?(Z) **then** Q_2 Z := ε; **end if**;
...
if B_n **and** nonem?(Z) **then** Q_n Z := ε; **end if**;
if nonem?(Z) **then** Q_{n+1} **end if**;

non $\mathbf{s} \in \Sigma^*$ edozein ikur eta Z, ez **P**-n ez kontrolerako beste makroegitura baten hedapenean erabili ez den makroaldagai berria diren.

Azkenik, sartutako makrobaldintzak eta makroaldagaiak eta **P** programak berak eduki litzakeen makroak hedatuz bukatu behar da.

4. PROPOSIZIOA: (predikatu errekurtsiboen eta funtzio konputagarrien bidez **kasuka definitutako** funtzioak ere konputagarriak dira beti) $\psi_1, \psi_2, ..., \psi_{n+1}: \Sigma^{*j} \longrightarrow \Sigma^*$ *n+1* funtzioak while-konputagarriak badira eta $P_1, P_2, ..., P_n \subseteq \Sigma^{*j}$ *n* predikatuak while-errekurtsiboak eta bateraezinak (hots, ez dira behin ere bi predikatu batera beteko) badira, orduan ondoren kasuka definitzen dugun $\xi: \Sigma^{*j} \longrightarrow \Sigma^*$ funtzioa ere while-konputagarria izango da:

$$\xi(x_1,...,x_j) = \begin{cases} \psi_1(x_1,...,x_j) & \text{baldin } P_1(x_1,...,x_j) \\ \psi_2(x_1,...,x_j) & \text{baldin } P_2(x_1,...,x_j) \\ \quad ... & \\ \psi_n(x_1,...,x_j) & \text{baldin } P_n(x_1,...,x_j) \\ \psi_{n+1}(x_1,...,x_j) & \text{baldin } \neg P_1(x_1,...,x_j) \wedge \neg P_2(x_1,...,x_j) \ ... \\ & \quad ... \wedge \neg P_n(x_1,...,x_j) \end{cases}$$

Frogapena

Aurrekotik zuzenean ondorioztatzen da. ξ funtzioa ondo definituta egoteko predikatuak bateraezinak izatea ezinbestekoa dela ohar zaitez.

3.4.2. case makroa

Noizean behin **case** motako baldintzazko agindu bat erabilgarria izango da, non kontrola baldintza arbitrarioen menpe egon ordez, aldagai baten balioen menpe dagoen. ADAn bere sintaxia **case V is when y_1 => Q_1 when y_2 => Q_2 .... when y_n => Q_n when others => Q_{n+1} end case;** da non V aldagai arrunt edo makroaldagai bat, $y_1,\ldots,y_n \in \Sigma^*$ eta $\mathbf{Q_1}$, ..., $\mathbf{Q_{n+1}}$ makroprogramak diren. Bere semantika honakoa da: V aldagaiaren balioa hartu eta $y_1,\ldots,y_n$ balioetako bat bada, dagokion $\mathbf{Q_1}$, ..., $\mathbf{Q_n}$ programa egikaritu; esandako balio horiekin bat ez baletor $\mathbf{Q_{n+1}}$ egikarituko litzateke. Makroaren sintaxia zailtasunik gabe aberas genezake, adibidez *when* klausulan balioak taldekatuz, baina ez dugu egingo aurrerago ez dugu beharko-eta.

case makroak dituen **P** makroprogramaren hedapena egiteko makroaren agerpen bakoitza era honetako makroprograma batengatik ordezkatuko dugu:

if V = y_1 **then** Q_1
elsif V = y_2 **then** Q_2
...
elsif V = y_n **then** Q_n
else Q_{n+1}
end if;

Ondoren **P**-k eduki litzakeen gainontzeko makroak (**if_then_else** makroegiturak, makrobaldintzak, makroadierazpenak eta makroaldagaiak) hedatu beharko dira.

12. ADIBIDEA: Demagun $\Sigma = \{\mathbf{a_1}, \ldots, \mathbf{a_n}\}$ alfabetoan ikurrak aurretik era honetan $\mathbf{a_1}<\mathbf{a_2}<...<\mathbf{a_n}$ ordenatuta daudela. Ondorioz Σ^*-ren hitzen artean eratortzen den ordena lexikografikoa har dezagun:

- $|x| < |y| \Rightarrow x < y$
- $\left.\begin{array}{l} |x| = |y| \\ i < j \end{array}\right\} \Rightarrow z\bullet\mathbf{a_i}\bullet x < z\bullet\mathbf{a_j}\bullet y$

Hau da, hitzak luzeraren arabera ordenatuta daude eta luzera berekoen artean ordena alfabetikoaren arabera. Alfabetoa, adibidez, {**a**,**b**,**c**} bada eta ikurren arteko ordena **a**<**b**<**c** dela hartzen badugu, hitzen ordena:

$\varepsilon < \mathbf{a} < \mathbf{b} < \mathbf{c} < \mathbf{aa} < \mathbf{ab} < \mathbf{ac} < \mathbf{ba} < \mathbf{bb} < \mathbf{bc} < \mathbf{ca} < \mathbf{cb} < \mathbf{cc} < \mathbf{aaa} < \mathbf{aab} < \mathbf{aac} <...$

Ordena hau osoa eta diskretua denez, *hur*: $\Sigma^* \rightarrow \Sigma^*$ funtzioa definitzen dugu, hitza emanda bere atzetik ordena horretan dagoena itzultzen duen funtzioa. Bere definizio induktiboa honakoa litzateke:

$$\mathrm{hur}(\varepsilon) = \mathbf{a_1}$$

$$\mathrm{hur}(w\bullet\mathbf{a_i}) = \begin{cases} w\bullet\mathbf{a_{i+1}} & \text{baldin } i < n \\ \mathrm{hur}(w)\bullet\mathbf{a_1} & \text{baldin } i = n \end{cases}$$

Funtzio hori while-konputagarria dela frogatuko dugu:

```
X0 := ε;
LAG := X1^R;
-- hur(LAG^R)•X0=hur(X1)
while car_an?(LAG) loop
      LAG := cdr(LAG);
      X0 := cons_a1(X0);
end loop;
```

if LAG = ε **then**
- - hur(ε)•X0=hur(X1)
    X0 := $cons_{a_1}$(X0);
else
- - hur(LAG^R)•X0=hur(X1) ∧ nonem?(LAG) ∧ ¬car_{a_n}?(LAG)
    Z := lehena(LAG);
    case Z **is**
    when 'a_1' => X0 := $cons_{a_2}$(X0);
    when 'a_2' => X0 := $cons_{a_3}$(X0);
    …
    when 'a_{n-1}' => X0 := $cons_{a_n}$(X0);
    end case;
    X0 := $(cdr(LAG))^R$ & X0;
end if;

Gainera *hur* funtzioa erabiliz while-konputagarriak diren funtzioen gainean beste emaitza bat froga dezakegu.

5. PROPOSIZIOA: While-konputagarria, osoa eta injektiboa den $f: \Sigma^* \to \Sigma^*$ funtzioa izanik, bere alderantzizko funtzioa $f^{-1}: \Sigma^* \multimap \Sigma^*$ while-konputagarria da ere.

Frogapena

f^{-1} funtzioak, injektiboa izango den arren, *osoa izateko beharrik ez duela* konturatzea garrantzitsua da. Noski, *w* hitza f-ren bidez inongo hitzaren irudia ez bada (ez da ran(f) multzokoa), orduan $f^{-1}(w)\uparrow$. Gure algoritmoak datu bezala *w* hitza hartzen du eta f(ε), f(hur(ε)), f(hur(hur(ε)), etab. hitzen berdina den egiaztatzen doa. Aurreirudia, f(y) = w betetzen duen *y*-ren bat aurkitzen badu, emaitza bezala itzultzen du; bestela ziklatu egiten du.

LAG := ε;
- - ∀Z(Z<LAG → f(Z)≠X1)
while f(LAG) /= X1 **loop**
    LAG := hur(LAG);
end loop;
X0 := LAG;

13. ADIBIDEA: Aipatu dugun ordena jarraituz hitz baten aurrekoa itzultzen duen *aurre:* $\Sigma^* \to \Sigma^*$ funtzioa while-konputagarria dela frogatzea orain erraza da. Hitzarmenez *aurre*(ε)=ε dela hartuko dugu, beraz ez da hur funtzioaren alderantzizkoaren berdin-berdina, baina predikatu errekurtsibo bat eta funtzio konputagarriak erabiliz kasuka defini dezakegu:

$$\text{aurre}(x) = \begin{cases} \varepsilon & \text{baldin } x = \varepsilon \\ \text{hur}^{-1}(x) & \text{bestela} \end{cases}$$

14. ADIBIDEA: ≤ predikatua while-errekurtsiboa dela frogatzea ere erraza da. Hori egiaztatzeko bere ezaugarri funtzioa honako makroprograma honek konputatzen duela ikustea nahikoa da:

```
    LAG1 := X1; LAG2 := X2;
-- X1≤LAG1 ∧ X2≤LAG2 ∧ (LAG1≤X2/LAG2≤X1)
    while LAG1 /= X2 and LAG2 /= X1 loop
        LAG1 := hur(LAG1);
        LAG2 := hur(LAG2);
    end loop;
    if LAG1 = X2 then X0 := 'a1'; else X0 := ε; end if;
```

Azkenik, ohikoak diren konparaketarako eragileak predikatu while-errekurtsiboak direla frogatzea begi-bistakoa da: > eragilea ≤-ren negazioa da, ≥ predikatua > eta = predikatuen disjuntzioa da eta ≥-ren ezeztapena eginez < lortzen da. Eragile hauek programetan, ohi bezala, ADAren notazioa jarraituz idatziko ditugu (<= eta >= ikurrak).

3.4.3. for makroa

Programazio lengoaietan ohikoa den beste agindu iteratibo bat *for* sententzia da, bere begiztak aldagai baten balioa banan-banan gehituz kontrolatzen dituena. Sintaktikoki **for V in A..B loop Q end loop;** idazten dugu, V *Q programak aldatzen ez duen* aldagai arrunt edo makroaldagai bat, A eta B aldagaiak, makroaldagaiak edo makroadierazpenak eta **Q** programa bat izanik. Makroaren esanahia: **Q** programa A..B tarteak adierazten duen hainbat aldiz egikaritzen da. Horretarako V aldagaia A balioarekin hasten da eta gehituz doa B balioa izan arte (*for*-a egikaritzen hastean adierazpenak zuen balioa). V-k hartutako balio bakoitzeko **Q** behin egikaritzen da. A balioa B baino handiagoa bada ez dela behin ere exekutatzen suposatzen dugu.

hur funtzioa eta ≤ predikatua erabiliz kontrolerako makroegitura honen hedapena defini dezakegu:

```
    V := A;
-- hurrengo esleipena beharrezkoa da Q-ren egikaritzapenak B-ren balioa aldatzen badu ere
    Z := B;
    while V ≤ Z loop
        Q
        V := hur(V);
```

end loop;

Era berean **for_reverse** makroa defini daiteke, bere sintaxia **for V in reverse A .. B loop Q end loop;** izanik. Kasu honetan V-k hasieran B balioa hartuko luke eta A balioa izan arte banan-banan gutxituz joango litzateke (A balioa B baino handiagoa balitz, makroa ez litzateke exekutatuko). Hedapena aurrekoaren berdintsua da.

3.5. Makroen erabilera

Ikusi dugun moduan makroprograma batean ezin dugu makroadierazpen bat sartu, bertan bere while-konputagarritasuna frogatu ez dugun funtzioren bat agertzen bada, eta era berean ezin dugu makrobaldintzarik erabili, bertan agertzen den predikatuaren while-errekurtsibitatea aurretik frogatu ez badugu. Apustua ez da txantxetakoa: uneren batean, frogatu gabe konputagarria dela suposatuta, funtzio bat erabiliko bagenu eta gero konputaezina suertatuko balitz, egindako programa eta horretan zuzenean zein zeharka oinarrituz egindako guztiak okerrak izango lirateke eta egindako erroreen kopurua, kalitatea eta posizioaren arabera eraikitzen saiatzen ari garen Konputagarritasun Teoriaren zati garrantzitsu bat baliogabetuko litzateke.

Horregatik, funtzioen while-konputagarritasuna eta predikatuen while-errekurtsibitatea frogatzeko makroprogramak erabiltzean hierarkiak osatuz lan egitea behar-beharrezkoa zaigu. Bereziki ekidin behar dugu:

- "Alderantziz" lan egitea, lehenik makroa erabili, bertan agertzen diren funtzioen edota predikatuen while-konputagarritasuna frogatzea gerorako utziz. Ordena faltak frogapenen bat "ahaztea" edo, okerragoa oraindik, frogapen zirkularrak egitea ekar lezake: agian ψ-ren konputagarritasuna χ-ren konputagarritasunean oinarritzen dugu, eta alderantziz ere, bakoitza bestearen programaren makroadierazpen batean agertzen delako.
- Azalez erraza dirudien funtzioa frogatu gabe "argi eta garbi konputagarri"tzat hartzea: argi eta garbi konputagarria bada, frogatzea ez da batere kostatuko.
- Itxuraz antzekoak eman arren, praktikan oso desberdinak izan daitezkeen funtzioak nahastea: kasu batzuetan bata while-konputagarria da eta bestea ez.
- Funtzioak, eta beraiekin bere konputagarritasuna, orokortzea. Demagun bi hitzen kateaketa while-konputagarria dela frogatzea ez dugula lortu, baina

bestalde hitz bat bere buruarekin kateatzen duen $f(x) = x \bullet x$ eragiketa konputagarria dela frogatzen duen programa aurkitu dugula. Hala ere, kateaketaren while-konputagarritasunari buruz ezer jakin gabe jarraitzen dugu. Are gehiago, Σ^*-ren w hitz bakoitzarentzat $g_w(x) = w \bullet x$ funtzioa while-konputagarria dela frogatuko bagenu ere, ez genuke kateaketaren while-konputagarritasunari buruz *ezer* frogatuko. Aurkakoa sinestea nahiko ohizkoa den okerra da, kontrako arrazonamendua zuzena delako: kateaketa while-konputagarria dela frogatuz, automatikoki f nahiz g_w funtzioak ere hala direla frogatzen dugu (**1.** proposizioan oinarritzen den partikularizazio prozesu bat izango litzateke).

- Kabiatutako adierazpenak lortzeko funtzio konputagarrien konposaketa konplexuak egitea lehendik prozesuaren zuzentasuna frogatu gabe.

Horregatik programa batetik bere makroetan erabiliko diren funtzioen eta predikatuen zerrenda ateratzea erabilgarria da, programaren dokumentazio ona bezala. Horrela makroen konputagarritasun edo errekurtsibitateari buruz arrazoitzea errazagoa izango da, bere programara nahiz **1.**tik **5.**era doazen proposizioen emaitzetara joz.

Jarraituko dugun estrategia honakoa da: denboran zehar bere while-konputagarritasuna frogatzen goazen funtzioak pakete batean (ADAn daudenen antzekoa) integratuta geratzen direla suposatuko dugu. Gure programak pakete hori "inporta" dezake bere funtzioak eta predikatuak erabil ahal izateko. Hau egiteko modua programaren hasieran pakete horren goiburukoa erazagutzea da, goiburukoarekin batera makroetan erabiliko diren funtzioen eta predikatuen (erazagupen mailan ez dira bereizten) zehaztapena jarriko da, beren argumentu kopurua azalduz. Honek ez du makroprograma gehiegi luzatuko, eta makroadierazpenen eta makrobaldintzen erabilera arautzeko balioko digu. Goiburukoa honelakoa izango da:

package KONPUTAGARRIAK **is**
    FG_1 FG_2 ... FG_n
end KONPUTAGARRIAK;

non FG_1, FG_2, ..., FG_n funtzio goiburukoak diren, makroetan erabilitako funtzioei dagozkienak. Beraz kasu bakoitzean paketeak programa idazteko behar ditugun funtzio while-konputagarriak taldekatzen ditu. Goiburuko horietako bakoitza era honetakoa da:

function fun (STRING, STRING, ..., STRING) **return** STRING;

fun: $\Sigma^{*j} \rightarrow \Sigma^*$ erabili nahi den funtzio while-konputagarria izanik, eta argumentu zerrendan STRING *j* aldiz errepikatuz. Makroprogrametan goiburukoak erabiltzearen inguruan zenbait zehaztasun egin behar dira:

- STRING, programetan Σ^* multzoari buruz aritzeko era da. *Datu-mota* bat adierazten du beraz, eta bere erabilera **4**. kapituluan justifikatuko dugu.
- ADAn ez bezala, funtzioen goiburukoetan ez dugu parametro formalen izena emango, kopurua eta mota bakarrik.
- Goiburukoan zuzenean konposaketa jarri ordez, makroan agertzen diren funtzio atomikoak adieraztea egokiagoa da. Adibidez, **LAG:=φ(ψ(X1,M,X4),χ(X4));** erako makroadierazpen bat badugu, φ (bi argumentu), ψ (hiru argumentu) eta χ (argumentu bat) funtzioentzat hiru goiburuko jartzea askoz egokiagoa da, denen emaitza den funtzioarentzat bakarra edukitzea baino, azken honek ziurrenik ez izenik ez esanahi garbirik ez duelako izango. Eragile logikoei buruz gauza bera esan daiteke.
- Predikatuak funtzioen antzera goiburukoan sartzen ditugu, azken finean horiek ere funtzioak baitira, heina $\{\mathbf{a_1}, \varepsilon\}$ duten funtzio osoak (heinari **4**. kapituluan beste zehaztasun bat ezarriko diogu).
- Goiburukoak notazio berezia duen (adibidez, infixu bidezko notazioa, esaterako *&* funtzioa eta konparaketa predikatuak, edo postfixu bidezko notazioa, ^{R} funtzioa bezala) funtzio bat daukanean hau komatxo artean idatziko da, ADAn egiten den moduan eragile bat datu-mota bat baino gehiagorekin erabili nahi denean.
- Gehiegi erabiltzen diren funtzioak eta predikatu sinpleenak ez ditugu behin eta berriro errepikatuko gure goiburuetan. Ekidingo ditugunen artean K_w funtzio konstanteak, $p_{j,k}$ proiekzioak, identitatea eta funtzio hutsa egongo dira
- Funtzio amankomunak erabiltzen dituzten makroprograma batzuk idatzi behar baditugu, guztientzat goiburuko bakarra erabil dezakegu.

15. ADIBIDEA: Honako makroprograma idatzi nahiko bagenu:

```
X0 := ε;
LAG := X1^R;
X0 := X3;
while cdr(LAG ) <= F(X0) or P(X1, LAG ) loop
      LAG := 'acc' & LAG ;
      X0 := Ψ(X0);
end loop;
```

if F(X0) <= X0 **then** X0 := ⊥⊥ ; **end if**;

honako erazagupena sartu beharko genuke:

```
package KONPUTAGARRIAK is
    function "R" (STRING) return STRING;
    function F (STRING) return STRING;
    function "<=" (STRING, STRING) return STRING;
    function P (STRING, STRING) return STRING;
    function "&" (STRING, STRING) return STRING;
    function Ψ (STRING) return STRING;
end KONPUTAGARRIAK ;
```

3.6. Kodeketa funtzioak

Aurreko ataletan zehar while programen adierazpen-ahalmena beste programazio lengoaia konbentzionalagoena adinakoa dela frogatzen lagundu diguten zenbait funtzio eta predikatuen while-konputagarritasuna eta while-errekurtsibitatea frogatu dugu. Gure lengoaiarekin programazio mekanismo arruntenak simulatzeko arazo gutxi izan dugun arren, oraindik ere ikutu ez dugun eta while programen eta beste lengoaien arteko desberdintasun nabarmenena sortzen duen arlo bat dago: programek erabiltzen dituzten objektu-moten arloa. Izan ere, alfabeto baten gaineko karaktere-kateak besterik ez ditugu onartzen. Itxuraz hutsunea den gai hau **4.** kapituluan ebatziko dugu, baina, hori baino lehen, kapitulu horretan baliagarriak izango diren zenbait funtzioen while-konputagarritasuna frogatuko dugu. Zehatzago esanda, hitz-segidak hitz bakarrean "konprimitu" edo "multzokatzen" dituzten **kodeketa funtzioak** defini ditzakegula ikusiko dugu.

Hasteko, $kod^2: \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$, hitz bikoteetarako kodeketa funtzioa, definituko dugu. Gure helburua hitz bikote bakoitza hitz bikotearen **kodea** deituko dugun hitz bakarrean kodetzeko eransketa-mekanismo bat edukitzea da, behar izanez gero kode horretatik abiatuta jatorrizko bikotea berreskuratzea zilegi izango delarik. Adibidez, kateaketa ez litzateke kodeketa funtzio egokia, izan ere (**abab**,**bb**) bikotea **ababbb** kodetuko litzateke eta **ababbb** kodetik abiatuta jatorrizko bikotea (**abab**,**bb**), (**ab**,**abbb**) edo kode hori banatzeko dauden gainerako bost aukeren artean zein den erabakitzea ezinezkoa litzateke. Hortik kod^2 funtzioa *injektiboa* izatea ezinbestekoa dela ondoriozta daiteke. Gainera supraiektiboa izateak ere lagunduko du, honela *hitz guztiak bikoteren baten kode* direla ziurtatuko dugulako (bestela, deskodetzerakoan erroreak sor litezke, eta oro har horrelakoak ekiditen saiatzen gara).

kod^2 funtzioa bijektiboa izatea komeni dela ikusi ondoren, funtzioa definitzeko Cantor-en metodoa erabiliko dugu. Orain ere **3.4.2** atalean definitu dugun edozein alfabetoren gaineko hitzen ordena har dezagun. Ordena osoa denez, hitz bakoitzari i ordinal bat egokitu eta w_i bezala adieraziko dugu. Ordinal txikiena 0 dela ulertuko dugu eta, beraz, induktiboki $w_0 = \varepsilon$ eta $w_{i+1} = \text{hur}(w_i)$ izango ditugu (ikus **1.** taula).

	$\Sigma = \{\mathbf{a}\}$	$\Sigma = \{\mathbf{a},\mathbf{b}\}$	$\Sigma = \{\mathbf{a},\mathbf{b},\mathbf{c}\}$
w_0	ε	ε	ε
w_1	**a**	**a**	**a**
w_2	**aa**	**b**	**b**
w_3	**aaa**	**aa**	**c**
w_4	**aaaa**	**ab**	**aa**
w_5	**aaaaa**	**ba**	**ab**
w_6	**aaaaaa**	**bb**	**ac**
w_7	**aaaaaaa**	**aaa**	**ba**
w_8	**aaaaaaaa**	**aab**	**bb**
w_9	**aaaaaaaaa**	**aba**	**bc**
w_{10}	**aaaaaaaaaa**	**abb**	**ca**

1. Taula: Bat, bi eta hiru ikur dituzten adibide-alfabetoetarako lehenengo 11 hitzen ordinalak.

Demagun bi dimentsiotako taula infinitua, bai lerro bai zutabeetan Σ^*-eko hitz ordenatuekin etiketatua badaukagula. Taularen sarrera bakoitza hitz bikote bati dagokio eta, beraz, *kod*^{2} funtzioaren definizio bat taula hori ordenan zeharkatu eta sarrera bakoitzari dagokion bikotearen kode-hitza egokitzearen baliokidea izango da. Sarrera bakoitza behin bakarrik bisitatuko da: lehendabizi w_0 ipiniko dugu (hala zein bikote kodetzen duen adieraziko delarik), ondoren w_1 eta horrela hitz guztiekin. Bide hori egiteko dauden era on guztietatik ondokoa aukeratuko dugu: goiko ezkerreko sarreratik hasiko gara, (ε, ε) bikoteari dagokionetik, eta ondoren, taularen diagonalak beteko ditugu, txikienetik hasita handienera, HM-IE norantzan, **2.** taulak adierazten duen bezala.

	$\mathbf{w_0}$	$\mathbf{w_1}$	$\mathbf{w_2}$	$\mathbf{w_3}$	$\mathbf{w_4}$	…	$\mathbf{W_j}$	…
$\mathbf{w_0}$	w_0	w_2	w_5	w_9	w_{14}	…		…
$\mathbf{w_1}$	w_1	w_4	w_8	w_{13}	w_{19}	…		…
$\mathbf{w_2}$	w_3	w_7	w_{12}	w_{18}	w_{25}	…		…
$\mathbf{w_3}$	w_6	w_{11}	w_{17}	w_{24}	w_{32}	…		…
$\mathbf{w_4}$	w_{10}	w_{16}	w_{23}	w_{31}	w_{40}	…		…
$\mathbf{w_5}$	w_{15}	w_{22}	w_{30}	w_{39}	w_{49}	…		…
$\mathbf{w_6}$	w_{21}	w_{29}	w_{38}	w_{48}	w_{59}	…		…
…	…	…	…	…	…	…		…
$\mathbf{w_i}$							$\mathbf{W_k}$	…
…	…	…	…	…	…	…	…	…

2. Taula: kod^2 funtzioaren definizioari dagokion eskema. Lerroak kodetu beharreko bikotearen lehen osagaia adierazten du, zutabeak bigarrena eta taularen sarrerak, berriz, kodea. Adibidez, (w_5,w_3) bikotearen kodea w_{39} izango da. {**a**,**b**} alfabetoaren kasuan (**ba**,**aa**) bikotearen kodeari **aaaba** hitza legokioke.

Eskema honi jarraituz, *kod*^{2} funtzioaren definizio induktiboa egin daiteke. Horretarako hiru kasu hartu beharko dugu kontuan: oinarrizkoa, diagonal berri baten hasiera (aurrekoaren bukaeraren arabera lortuko dena) eta gainerako egoerak:

- $\text{kod}^2(\varepsilon, \varepsilon) = \varepsilon$
- $\text{kod}^2(\text{hur}(w), \varepsilon) = \text{hur}(\text{kod}^2(\varepsilon, w))$
- $\text{kod}^2(v, \text{hur}(w)) = \text{hur}(\text{kod}^2(\text{hur}(v), w))$

Taula betetzeko erabili dugun metodoagatik *kod*^{2} funtzioa *osoa* eta *bijektiboa* dela ziurta dezakegu: osoa, posizio guztiak noizbait betetzen direlako (hitz bikote bakoitzari kode bat dagokio); injektiboa, ordenan egiten denez, hitz bera posizio desberdinei esleitzea ezinezkoa delako (bikote desberdinek kode desberdina dute); supraiektiboa, hitz guztiak momenturen batean posizioren batean kokatu direlako (hitz guztiak bikoteren baten kode dira).

Azkenik, funtzioa *while-konputagarria* da, taula betetzeko erabilitako ideia berari jarraituz programa bat eraiki dezakegulako: (w_i, w_j) bikote baten kodea kalkulatzeko (w_0, w_0) bikotearen kodea kalkulatuz hasiko gara, gero (w_1, w_0), (w_0, w_1), (w_2, w_0), (w_1, w_1), (w_0, w_2), eta horrela, taularen diagonalak ordenan jarraituz, sarrerako bikotearen kodea sortu arte jarraituko genuke. Aurreko

kapituluan esan bezala, dagokion makroprogramak, erabiliko dituen funtzio konputagarri guztien erazagupenak biltzen dituen goiburukoa beharko du:

```
package KONPUTAGARRIAK is
        function "/=" (STRING, STRING) return STRING;
        function "=" (STRING, STRING) return STRING;
        function hur (STRING) return STRING;
        function aurre (STRING) return STRING;
end KONPUTAGARRIAK ;

LERROA := ε;
ZUTABEA := ε;
X0 := ε;
- - X0 = kod2(LERROA, ZUTABEA)
while LERROA /= X1 or ZUTABEA /= X2 loop
        if LERROA = ε then
- - Kasu honetan diagonalaren bukaerara iritsi gara eta hurrengoa hasi behar dugu
                LERROA := hur(ZUTABEA);
                ZUTABEA := ε;
        else
- - Kasu honetan diagonal berean jarraitzen dugu
                LERROA := aurre(LERROA);
                ZUTABEA := hur(ZUTABEA);
        end if;
        X0 := hur(X0);
end loop;
```

kod^2 funtzioaren while-konputagarritasuna programa honi esker zalantzan jar ez daitekeen arren, erabilitako algoritmoa ez da oso praktikoa, kode zehatz bat kalkulatu nahi dugunean jakinminak jota besterik ez bada ere. Modu zuzenago bat topatzeko ondokoa egingo dugu:

- Diagonalak txikienetik handienera zenbatuko ditugu, 0.a (w_0, w_0) bikotearen gelaxka besterik ez duena izango delarik.
- k. diagonalak, beraz, *k+1* osagai izango du.
- k. diagonalean dauden gelaxkei dagozkien bikoteak (w_i, w_j) erakoak dira, $i+j = k$ izanik. Diagonal bakoitzaren lehenengoa (w_k, w_0) bikotea da.
- *(i+j).* diagonalean dagoen (w_i, w_j) bikotearen w_k kodearen azpindizea kalkulatzeko, lehendabizi aurreko diagonalek dituzten osagai kopuruak batu behar dira (*0.*etik *(i+j-1).*era), eta ondoren bere diagonalean aurretik duen osagai kopurua gehitu behar da (bere posizioa ez da batu behar, w_0 kodeari dagokiona zenbatu delako).

Horrela, *k* kalkulatzeko (zenbakizko) modu azkarragoa honakoa litzateke:

$$k = \frac{(i+j) * (i+j+1)}{2} + j$$

Bestalde, kod^2 funtzio bijektiboa denez, alderantzizko bi **deskodeketa** funtzio defini ditzakegu, $deskod_{2,1}$, $deskod_{2,2}$: $\Sigma^* \rightarrow \Sigma^*$, kodetik abiatuta jatorrizko bi osagaiak berreskuratzeko erabiliko ditugunak.

Ondoko ekuazioa egiazkoa egitea lortu nahi dugu:

$$kod^2 (deskod_{2,1} (w), deskod_{2,2} (w)) = w$$

Funtzio hauek ere osoak, supraiektiboak eta while-konputagarriak dira. Konputatzeko programa kod^2 funtzioarenaren antzera eraikitzen da, baina oraingo honetan kodetik abiatzen garela kontuan izanik. Ideia jarraian dauden bikoteak sortzen joatean datza, eta zuzena topatzen dugunean dagokion balioa itzuliko dugu: lerroa ($deskod_{2,1}$-en kasuan) edo zutabea ($deskod_{2,2}$-ren kasuan):

```
package KONPUTAGARRIAK is
    function "/=" (STRING, STRING) return STRING;
    function "=" (STRING, STRING) return STRING;
    function hur (STRING) return STRING;
    function aurre (STRING) return STRING;
end KONPUTAGARRIAK;

LERROA := ε; ZUTABEA := ε; LAG := ε;
- - LAG = kod2(LERROA, ZUTABEA)
while LAG /= X1 loop
    if LERROA = ε then  LERROA := hur(ZUTABEA);      ZUTABEA :=
    ε;
    else   LERROA := aurre(LERROA);
           ZUTABEA := hur(ZUTABEA);
    end if;
    LAG := hur(LAG);
end loop;
- - deskod2,1-entzat soilik. deskod2,2-rentzat azken agindu hau X0:=ZUTABEA; izango da
X0 := LERROA;
```

Bi hitz bakarrean kodetzen dakigunetik, edozein hitz kopururentzat kodeketa bila dezakegu kod^2 funtzioa behin eta berriro aplikatuz: zein ordenatan aplikatu behar den besterik ez dugu kontuan izan beharko. Edozein $k \geq 1$ izanik, *k* bakoitzeko kodeketa funtzio bat definituko dugu, kod^k: $\Sigma^{*k} \rightarrow \Sigma^*$ argumentu kopuruaren gainean induktiboki honela definituz:

- $kod^1(z) = z$

- $kod^{k+1}(z_1, \dots, z_{k+1}) = kod^2(z_1, kod^k(z_2, \dots, z_{k+1}))$

Hau da, edozein tupla kodetzeko kod^2 funtzioa behin eta berriz aplikatuko dugu azkeneko bi hitzetatik hasita:

$$kod^k(z_1, z_2, \dots z_{k-1}, z_k) = kod^2(z_1, kod^2(z_2, \dots kod^2(z_{k-1}, z_k)\dots))$$

kod^{k+1} erako kodeketa funtzio guztiak osoak, bijektiboak eta while-konputagarriak izango dira dagokien kod^k funtzioa hala bada (izan ere, konposaketaren gainerako osagaientzat, hots, kod^2 eta *id*-entzat, dagoeneko frogaturik dago). kod^1-ek hiru baldintzak betetzen dituenez, funtzio-multzo guztiaren osotasuna, bijektibotasuna eta while-konputagarritasuna betetzen direla induktiboki froga dezakegu.

Kasu honetan ere deskodeketa funtzioak defini ditzakegu, $deskod_{k,i}: \Sigma^* \to \Sigma^*$erakoak. Funtzio hauek hitz batek kodetzen duen k-tuplaren i. osagaia itzuliko digute eta hauek ere osoak, supraiektiboak eta while-konputagarriak izango dira honela defini ditzakegulako:

$$deskod_{k,i}(w) = \begin{cases} w & \text{baldin } i=1 \wedge k=1 \\ deskod_{2,1}(w) & \text{baldin } i=1 \wedge k>1 \\ deskod_{k-1,i-1}(deskod_{2,2}(w)) & \text{baldin } i>1 \wedge k>1 \end{cases}$$

eta berriz ere induktiboki honakoa arrazoi daiteke:

- $deskod_{k,1}$ erako funtzio guztiak while-konputagarriak, osoak eta supraiektiboak dira (*id* eta $deskod_{2,1}$ hala direlako)
- *j*-ren balio baterako $deskod_{k,j}$ erako funtzio guztiak while-konputagarriak, osoak eta supraiektiboak badira, orduan $deskod_{k,j+1}$ erako guztiak ere hala dira ($deskod_{2,2}$ hala delako)

16. ADIBIDEA: Tupla zehatz bati funtzio hauek nola aplikatu ikusiko dugu. Demagun $\mathbf{kod^4(w_5,w_2,w_0,w_1)}$ eta $\mathbf{deskod_{4,3}(w_{178})}$ kalkulatu nahi ditugula. Definizioak jarraituz:

$$\begin{aligned} kod^4(w_5,w_2,w_0,w_1) &= kod^2(w_5,kod^3(w_2,w_0,w_1)) = \\ &= kod^2(w_5,kod^2(w_2,kod^2(w_0,w_1))) = kod^2(w_5,kod^2(w_2,w_2)) = \\ &= kod^2(w_5,w_{12}) = w_{165} \end{aligned}$$

$$\begin{aligned} deskod_{4,3}(w_{178}) &= deskod_{3,2}(deskod_{2,2}(w_{178})) = deskod_{3,2}(w_7) = \\ &= deskod_{2,1}(deskod_{2,2}(w_7)) = deskod_{2,1}(w_1) = w_1 \end{aligned}$$

4. Datu-motak

Makroen erabilerari esker, gure lengoaiaren sintaxi mugatuak sortutako arazoak gainditzea lortu dugu: kodea errepikatu beharra, aldagaien erabileraren murriztapenak edota oinarrizko kontrol-egitura eta funtzioen multzo onargarriaren eza. Honek lengoaia konbentzionalek dituzten kontrol-osagai gehientsuenak while programen bidez simula ditzakegula erakutsi digu, eta horrela, gure hasierako lengoaia batere aldatu gabe, laburdura (makroprograma) gisa gehitu ditugu. Egia esan, guk egin duguna ez dago mundu errealean gertatzen denetik oso urruti. Konputagailu baten PUZak (hauxe baita benetan gauzak "egiten" dituena) eragiketa-multzo nahiko txikia onartzen du, zenbait arkitekturetan benetan oso txikia, eta horren gainean erabiltzailearengan makina askoz konplexu eta erosoagoa delako irudipena sortzen duen interfazea programatzen da. Baina benetan alegiazko makina horren arabera lan egin beharrean, PUZaren gainean zuzenean egin daiteke (eta berez horixe da interfazeak egiten duen itzulpena).

Hala ere, kontrolaren osagarria den alderdi batean ez dugu batere aurreratu: gure while programek erabil ditzaketen datu-motak erabat mugatuak daude. Zenbakizko objektuak edo objektu logikoak edota bektore edo fitxategi gisako agregazioak onartzen ez dituen programazio lengoaia batek ez dirudi oso errealista denik. Zergatik mugatu gara STRING datu-motara? Oinarrizko bi arrazoi hauengatik:

- Lehenengoa, horixe delako konputagailu errealetan gertatzen dena. Erabiltzailearen ikuspegitik badirudi objektu-mota desberdin asko dagoela, baina hori irudipen hutsa da, kanpotik, ordenadore batean datuak sartu edo ateratzeko ondo zehaztutako alfabeto baten (ASCII adibidez) gaineko karaktere-sekuentzien bidez egiten dugulako. Barrutik, aldiz, konputagailuak datu horiek jaso eta bere alfabetora (bitarrera adibidez) itzultzen ditu eta, karaktere-kateei dagozkien eragileen bidez, lortutako sekuentzien gainean lan egiten duelako.
- Bigarrena, ez dugulako hemen bukatuko. Makroekin bezala, gure lengoaiak ez dituen **datu-moten** konputazioa simulatzen saiatuko gara. Horregatik, STRING-en aukeraketa $cons_S$ eta *cdr* funtzioena bezain baliagarria izango da. STRING gainerako datu-motak simulatzeko oinarri egokia dela eta, beraz, lengoaia minimoa lortzeko helburua betetzen dela egiaztatuko dugu.

Informatikak erakusten digunaren arabera, eragiketen abstrakzio eta modulartasunerako mekanismoak oro har datuen mailari dagozkionak baino askoz ere errazagoak dira. Horregatik, datu-motak simulatzeko erabiliko dugun laburdura-sistemak makroetan ikusitakoarekiko zenbait desberdintasun tekniko izango du, baina filosofia berean datza:

- lehendabizi, while programek ez duten baina edukitzea interesatzen zaigun datu-mota bat zehazki identifikatu eta definituko dugu
- ondoren, lengoaiaren osagaia izan ez arren, while programen elementuen bidez simula daitekeela egiaztatuko dugu
- hortik aurrera *lengoaiaren osagaia bailitz* erabiliko dugu.

14. DEFINIZIOA: **T** datu-mota egitura algebraikoa da, bere osagaiak honakoak izanik, **datu-multzo** bat edo motaren balio posibleen multzoa (oraingoz **T** deituko duguna) eta $\{\psi_1, \psi_2, \ldots, \psi_n\}$ funtzio-multzoa; ψ_i bakoitza motaren **eragiketa** deitzen da, eta ψ_i: $T_1 \times \ldots \times T_j \rightarrow T_{j+1}$ erakoa da, non T_k motetariko bat gutxienez **T** bera den. Mota bat definitzen duten eragiketek oro har mota horretako argumentuak eta emaitzak izango dituzte, baina eragiketa mistoak ere izan daitezke, non argumentuetariko batzuk edota emaitza beste mota bati dagozkion (predikatuen kasuan hala gertatuko da).

Hortaz, gure programazio lengoaiari datu-mota baten ezaugarriak gehitu nahi badizkiogu, bi arazo ebatzi beharko dugu: bere datuak simulatzea eta bere eragiketak simulatzea. Adibidez, mota boolearraren kasuan, lehendabizi bere balioak {true, false} dauzkagun baliabideen bitartez adierazi beharko ditugu, hots, alfabeto baten gaineko hitzen bitartez. Alfabetoa {**0**, **1**} bada, *false* '**0**' hitza eta *true* '**1**' hitza izatea erabaki dezakegu. Alfabetoa {**a**, **b**, ..., **z**} bada, *true* '**egiazkoa**' eta *false* '**faltsua**' hitzen bidez adieraztea hobes dezakegu. Horrela, while programen aldagaietan balio horiekin benetan balio boolearrak *bailira* lan egin ahal izango dugu, modu horretan *interpreta* ditzakegulako. Berez, predikatu while-errekurtsiboak definitu genituenean ideia honen aurrerapena egin genuen: balio boolearrik erabili ezin genuenez, '$\mathbf{a_1}$' hitza *true* eta hitz hutsa *false* bezala interpretatzea erabaki genuen, eta horrek, mota logikorik izan gabe, baldintzazko agindu zein iteratiboen kontrola bideratzeko eragiketa logikoekin lan egiten utzi digu.

Objektuak simulatzeko moduren bat aurkitzen dugunean, horien gainean lan egiten duten eragiketekin gauza bera egin beharko dugu, baina lehenengoa lortzean bigarrenaren ebazpidea zuzenean lortuko da. Balio boolearren adibidearekin jarraituz, disjuntzio eragiketaren kasuan, gure lengoaian eragiketa

hori egin daitekeen konprobatu beharko genuke. Horretarako, balio boolearrak adierazten duten bi hitz ('**faltsua**' eta '**egiazkoa**' esaterako) jaso eta bere disjuntzioa adieraziko duen hitza (kasu honetan '**egiazkoa**') itzuliko duen while programarik dagoen konprobatu beharko dugu.

Prozesu hau datu-motaren **inplementazioa** deitzen da, eta hurrengo atalean zehazkiago deskribatuko dugu.

4.1. Inplementazioak

Oro har **T** datu-mota inplementatzeak, inplementatu nahi den motako (**mota abstraktua**) datuak inplementaziorako erabiliko dugun motako (**mota konkretua**) objektuen arabera nola adieraziko ditugun erabakitzea esan nahi du. Gure kasuan, STRING datu-mota besterik ez dugunez, horixe izango da (oraingoz behintzat) edozein inplementaziotan erabiliko dugun mota konkretua. Mota bietako datuen arteko erlazioa (hitzen eta inplementatu beharreko objektuen artekoa) bi eratara ezar daiteke:

- adierazpide-mekanismoa emanez, hots, datu-mota berriaren balio bat izanik, bera adieraziko duen hitza zein izango den esanda
- interpretazio-mekanismoa emanez, hots, hitz bat izanik, datu-mota berriaren zein baliori dagokion esanda.

Bigarren mekanismo honetan jarriko dugu arreta, inplementazioan eragin handiena honek izango baitu.

15. DEFINIZIOA: Bedi **T** datu-mota.

$$\Im_{\mathbf{T}}: \Sigma^* \rightarrow \mathbf{T}$$

erako edozein funtzio oso eta supraiektibo, **interpretazio- edo abstrakzio-funtzio** deituko dugu. Funtzio honek, bere izenak adierazten duen legez, Σ^*-eko hitz bakoitza, **T**-ko osagai bat adierazten duela suposatuz, nola interpretatu behar den ezarriko du.

$\Im_{\mathbf{T}}$-k bete behar dituen baldintzak honakoak dira:

- *Supraiektiboa* izatea, datu-motako objektu guztiak gutxienez hitz batez adierazi ahal izateko. Bestela, ez ginateke **T** mota inplementatzen ariko, bere azpimultzo bat baizik.
- *Osoa* izatea, behar izanez gero Σ^*-eko edozein hitz **T**-ko osagai gisa interpretatu ahal izateko. Baldintza hau eskatzea erabaki metodologikoa da,

izan ere, zenbait hitzek daturik ez adieraztea onar genezakeen arren, segurtasunari begira ez litzateke komenigarria. Azken finean, gure programek **T** motako objektuak maneiatuko bailituzte lan egiten dugun arren, programek benetan ikur-kateak prozesatuko dituzte, ondoren $\Im_{\mathbf{T}}$-ren arabera objektu gisa interpretatuko ditugularik. Prozesu honetan **T**-n esanahirik ez duen hitzik sortuko balitz, erroreak sortuko lirateke eta hauek ekidin nahi ditugu. Boolearren kasuan, *true* '**1**'ekin eta *false* '**0**'rekin adierazteko erabakiak ez du baliorik izango ez badiegu Σ^*-eko gainerako hitz guztiei esanahirik ematen. Adibidez, **1**ez hasten diren hitz guztiei *true* eta gainerakoei (ε barne) *false* esanahia ematea erabaki dezakegu.

- *Injektibotasuna* ez da beharrezkoa, objektu batentzat adierazpide bat baino gehiago izan dezakegulako. Hala ere, $\Im_{\mathbf{T}}$-ren hautazko injektibotasuna (eta beraz bijektibotasuna) gogoan izango dugun ezaugarria da.

Hemen esandakoak, interpretazio-funtzioak definitzeko garaian kardinaltasun-murriztapenak garrantzitsuak direlako ondoriora garamatza. **T** mota finitua bada, ezinezkoa da bijektibo izatea, Σ^*-en kardinaltasuna infinitua baita. **T** infinitu zenbagarria bada, bijektibo izatea ala ez izatea aukera dezakegu. Eta **T** ez bada zenbagarria, orduan interpretazioa ezinezkoa da. Honen ondorioren bat aski ezaguna da Informatikan: zenbaki errealak ezin dira programazio lengoaietan inplemetatu. Hauen ordez, FLOAT bezalako motak erabiltzen dira. Berez, doitasun finkoa duten hurbilpenak dituzte eta hauen ezaugarriak eta zenbaki errealenak hain dira desberdinak ezen hauek definitzeko eta ondorioak aztertzeko disziplina bat (Zenbakizko Analisia) sortu behar izan baita.

Hitzetik adierazitako datura pasatzeko funtzioa definitu dugunean, alderantzizko mekanismoa edukitzea ere interesgarria izango da askotan. Mekanismo horri **adierazpide-funtzio** deituko zaio:

$$\Re_{\mathbf{T}}: \mathbf{T} \rightarrow \Sigma^*$$

eta datu bat izanik adieraziko duen hitzetako bat (bere **adierazpide kanonikoa**) emango digu. Oro har $\Im_{\mathbf{T}}$ diseinatu aurretik nolako $\Re_{\mathbf{T}}$ nahi dugun pentsatzea lagungarri izango da (izan ere hau funtzio "naturalagoa" da).

Interpretazio-funtzioa bijektiboa denean, $\Re_{\mathbf{T}} = \Im_{\mathbf{T}}^{-1}$ beti betetzen da. Bestela, adierazpide-funtzioak $\Im_{\mathbf{T}}(\Re_{\mathbf{T}}(x)) = x$ bete beharko du **T**-ko edozein *x* osagai izanik. Begi-bistakoa da interpretazio-funtzioa bakarra ez dela, baina behin definitu dugunean inplementatzen ditugun eragiketak definizioari jarraituz simulatu beharko ditugu.

Eragiketen simulaziorako mekanismoa atal honekin hasi aurretik deskribatu dena bezalakoa da. Zenbait hitzek bere horretan direnaren esanahi desberdina adieraziko balute bezala lan egin behar badugu, horrekin bat etorriko diren programak diseinatu beharko ditugu: programetan hitz horiek maneiatzean lortutako emaitzek euren atzean ikusi nahi dugun hori adierazten jarraitu beharko dute.

16. DEFINIZIOA: Bedi ψ: $T_1 \times \ldots \times T_j \longrightarrow T_{j+1}$ **T** motaren gaineko eragiketa (hots, T_k-etariko bat gutxienez **T** bera da). Bitez $\Im_{\mathbf{T}_1}$, $\Im_{\mathbf{T}_2}$, ..., $\Im_{\mathbf{T}_{j+1}}$ ψ-ren definizioan agertzen diren datu-motetarako interpretazio-funtzioak. **P** while programak **ψ inplementatzen** duela esango dugu edozein z_1, z_2, ..., z_j hitzetarako honakoa betetzen badu:

a) baldin $\psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), ..., \Im_{\mathbf{T}_j}(z_j))\uparrow$ orduan $\varphi_{\mathbf{P}}^{j}\ (z_1, z_2, ..., z_j)\uparrow$

b) baldin $\psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), ..., \Im_{\mathbf{T}_j}(z_j))\downarrow$ orduan $\varphi_{\mathbf{P}}^{j}\ (z_1, z_2, ..., z_j)\downarrow$ eta gainera

$$\Im_{\mathbf{T}_{j+1}}(\varphi^{j}{}_{\mathbf{P}}\ (z_1, z_2, ..., z_j)) \simeq \psi(\Im_{\mathbf{T}_1}(z_1), \Im_{\mathbf{T}_2}(z_2), ..., \Im_{\mathbf{T}_j}(z_j))$$

$$\begin{array}{ccc} T_1 \times T_2 \times \ \ldots \ \times T_j & \xrightarrow{\ \psi\ } & T_{j+1} \\ \uparrow & & \uparrow \\ \Im_{T_1} \times \Im_{T_2} \times \ \ldots \ \times \Im_{T_j} & & \Im_{T_{j+1}} \\ \uparrow & & \uparrow \\ \Sigma^* \times \Sigma^* \times \ \ldots \ \times \Sigma^* & \xrightarrow[\varphi_P^j]{} & \Sigma^* \end{array}$$

Hau da, **P** while programak z_1, z_2, ..., z_j *j* hitz hartzen ditu argumentu gisa eta *w* beste hitz bat itzultzen du emaitza gisa, non *w*-k adierazten duen balioa z_1, z_2, ..., z_j-k adierazten duten balioen gainean ψ aplikatzean lortzen den balio bera den. Era berean, zenbait balio adierazten dituzten hitzen gainean **P** dibergentea izango da, balio horietarako ψ definitu gabe dagoenean. Orduan interpretazioak ψ-rekiko homomorfoak direla esango da, aurreko trukakortasun-eskema betetzen delako.

17. DEFINIZIOA: Bedi (**T**, $\{\psi_1, \psi_2, \ldots, \psi_n\}$) datu-mota. **T**-ren implementazioa ($\Im_{\mathbf{T}}$, $\{P_1, P_2, ..., P_n\}$) bikotea da, non $\Im_{\mathbf{T}}$ **T**-ren datuetarako interpretazio-funtzioa

den eta P_i erako programa bakoitzak ψ_i funtzioa inplementatzen duen. $\Im_{\mathbf{T}}$ funtzioa bijektiboa bada, inplementazioa **hertsia** dela esango da.

Beraz, (**T**, $\{\psi_1, \psi_2, \ldots, \psi_n\}$) datu-mota inplementatzeko ondoko pausoak eman behar ditugu:

- $\Im_{\mathbf{T}}: \Sigma^* \rightarrow \mathbf{T}$ interpretazio-funtzioa aurkitu. Bijektiboa ez denean, dagokion $\Re_{\mathbf{T}}: \mathbf{T} \rightarrow \Sigma^*$ adierazpide-funtzioa ere definitu beharko da.
- $\psi_1, \psi_2, \ldots, \psi_n$ funtzioen definizioetan agertzen diren datu-mota guztien zerrenda egin (**T** eta STRING motak ezik). Mota hauek **T**-ren inplementazioarekin hasi aurretik inplementatuak egotea ezinbestekoa da.
- Motaren eragiketak inplementatzen dituzten $\{P_1, P_2, ..., P_n\}$ programak eraiki. Programa hauen goiburukoa (oro har guztiek amankomunean izango dutena) honelakoa izango da:

  package KONPUTAGARRIAK **is**
  MG_1 MG_2 ... MG_m
  FG_1 FG_2 ... FG_p
  end KONPUTAGARRIAK;

  non MG_1, MG_2, ..., MG_m aurreko puntuan aipatutako eta aldez aurretik inplementatuak egon behar duten motei dagozkien goiburukoak diren, eta FG_1, FG_2, ..., FG_p aurreko kapituluan deskribatu ziren bezala definitutako funtzio-goiburukoak diren. Mota-goiburuko bakoitza honelakoa izango da:

  type T_i **is** STRING;

  honela, motaren osagaiak zuzenean Σ^*-eko hitzekin simulatzen direla adieraziko da.

4.2. Oinarrizko moten inplementazioa

Ondoren, konputazio logikoak eta zenbakizkoak egiteko erabiliko ditugun oinarrizko bi moten inplementazioen adibideak ikusiko ditugu. Oro har interpretazio-funtzioak ahalik eta modu errazenean diseinatzen saiatuko gara, beti ere ondoko bi ezaugarriak betetzen saiatuz:

- bijektiboak izatea. Inplementazio hertsiak aukeratzeak bi arrazoi ditu: batetik, ez dugu adierazpide-funtzioa definitu behar, eta, bestetik, inplementatutako motaren balio bakoitzeko adierazpide bakarra dugunean, eragiketak inplementatzea oro har errazagoa gertatzen da.

- alfabetoarekiko independentea izatea. Informatikan erabili ohi diren inplementazioak alfabeto bitarrerako eta datuak antolatzeko modu zehatz baterako pentsatu dira; gure aldetik, edozein egoeratan aplikagarriak izatea nahiago dugu, lortutako soluzioak zenbaitetan "antinaturalak" iruditu arren.

Bestalde, mota bakoitzeko eragiketak kontu handiz aukeratzen saiatuko gara, ahal den multzorik txikiena hartzen saiatuz beti. Aurrerago ikusiko dugunez, honen arrazoia hauxe da: behin inplementaturik, mota bere eragiketa-multzoa handitzeko murriztapenik gabe erabili ahal izango dela. Hortaz, hasiera batean behar-beharrezkoak direnak definituko ditugu, eta ondoren mota nahi dugun haina aberastuko dugu.

4.2.1. Mota BOOLEARRA

Eragiketa logikoak inplementatzeko sortu den datu-mota hau **B** = {true, false} multzoak eta ukapen eta berdintasun (baliokidetza logikoa) eragiketek osaturik dago.

$$\neg: \mathbf{B} \rightarrow \mathbf{B}$$

$$<->: \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$$

Erabiliko dugun interpretazio-funtzioa ezaugarri-funtzioak konputatzeko gure ohituran oinarriturik dago: *false* konstantea hitz hutsarekin adieraziko dugu eta *true* konstantea $\mathbf{a_1}$ hitzarekin, non $\mathbf{a_1} \in \Sigma$ alfabetoaren lehen osagaia den. Beste edozein hitz ere *true* bezala interpretatuko dugu. Adierazpide honen abantaila (LISP programazio lengoaian ezik informatikan oro har erabiltzen denaren oso bestelako adierazpidea dena) erabiltzen den alfabetoarekiko independentea izatea da.

$$\Im_{\mathbf{B}}(w) = \begin{cases} \text{false} & \text{baldin } w = \varepsilon \\ \text{true} & \text{bestela} \end{cases}$$

Logikoki, interpretazio hau ez da bijektiboa, beraz, adierazpide-funtzioa ere definitu behar dugu:

$$\Re_{\mathbf{B}}(t) = \begin{cases} \varepsilon & \text{baldin } t = \text{false} \\ \mathbf{a_1} & \text{baldin } t = \text{true} \end{cases}$$

Lehendabizi argumentu bakarreko ukapen funtzioa inplementatuko dugu, bere emaitza argumentuaren balioaren aurkakoa izanik:

```
package KONPUTAGARRIAK is
end KONPUTAGARRIAK;
```

X0 := ε;
if not nonem?(X1) **then** X0 := $cons_{a_1}$(X0); **end if**;

Eta bukatzeko, berdintasuna. Interpretazio bijektiboa ez denez, balioen berdintasuna ez da hitzen berdintasunaren bidez adierazten, lehenengoa bere bi argumentuek balio bera *adierazten* dutenean betetzen delako. Ezaugarri hau inplementazio ez-hertsietan agertu ohi da, eta berdintasuna programatu behar izaten da. Horregatik ezin dugu boolearren arteko berdintasuna "=" ohiko ikurraz adierazi, bestela, gure programetan erabiltzean, hitzen arteko berdintasunarekin nahasiko baikenuke.

package KONPUTAGARRIAK **is**
end KONPUTAGARRIAK;

X0 := ε;
if (**not** nonem?(X1) **and not** nonem?(X2) **or**
  (nonem?(X1) **and** nonem?(X2)) **then**
    X0 := $cons_{a_1}$(X0);
end if;

Boolearrak definitu ditugunez, gure predikatuak eraikitzeko erabili ahal izango ditugu. Hemendik aurrera predikatuak emaitza BOOLEARRA duten funtzio gisa definituko ditugu. Gauza bera esan dezakegu multzoen ezaugarri-funtzioei dagokienean ere. Berez, orain arte definitutako ezaugarri-funtzio eta predikatu guztiak emaitza BOOLEARRA duten funtzioak bihurtuko ditugu.

4.2.2. Mota ARRUNTA

Datu-mota hau $\mathbf{N} = \{0, 1, 2, 3, \ldots\}$ zenbaki arrunten multzoak eta hurrengo (aurgumentu bakarrekoa) eta berdintasun (predikatu bitarra) eragiketek osaturik dago:

$$\text{suc}: \mathbf{N} \rightarrow \mathbf{N}$$

$$=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

Interpretazio-funtzioa definitzeko zenbaki arrunten ordenari edozein alfabetoren gaineko hitzetarako **3.4.2** atalean definitutako ordena egokituko diogu, honela, hitz hutsak 0a adieraziko du, eta *w* hitzak *k* zenbakia adierazten badu, orduan *hur(w)* hitzak *suc(k)* adieraziko du. Hortaz, $\Im_{\mathbf{N}}(w_k) = k$ beteko da.

Baldin $\Sigma = \{ \mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n} \}$, orduan $\Im_{\mathbf{N}}: \Sigma^* \rightarrow \mathbf{N}$ funtzioa induktiboki honela definituko dugu:

$$\Im_{\mathbf{N}}(\varepsilon) = 0$$

$$\Im_{\mathbf{N}}(x \bullet \mathbf{a_i}) = n * \Im_{\mathbf{N}}(x) + i$$

Interpretazioa horrela definituta, *suc* funtzioa *hur* funtzioaren parekoa da. Beraz, honako programa errazak konputatuko luke:

```
package KONPUTAGARRIAK is
    function hur (STRING) return STRING;
end KONPUTAGARRIAK;

X0 := hur(X1);
```

Berdintasunarekin beste horrenbeste gertatuko da. Inplementazio hertsia denez, bi zenbaki arrunt berdinak izango dira baldin eta soilik baldin hitz berak adierazten baditu. Hortaz, zenbaki arrunten arteko berdintasun-predikatua konputatzen duen programa honakoa izango da:

```
package KONPUTAGARRIAK is
    type BOOLEARRA is STRING;
    function "=" (STRING, STRING) return BOOLEARRA;
end KONPUTAGARRIAK;

X0 := X1 = X2;
```

Kasu honetan, zenbaki arrunten arteko berdintasuna eta hitzen artekoa ikur beraz adierazteak ez du arazorik sortuko. Ohar zaitez hitzen arteko berdintasun-predikatu while-errekurtsiboa emaitza boolearra duen funtzio gisa lehenengo aldiz erabili dugula (makrobaldintza batean erabili beharrean makroadierazpen batean).

Oro har, aurreko kasuen antzera funtzio berri bat definitzean aurretik programatutako baten baliokidea dela ikustean, aipatu besterik ez dugu egingo, eta ez dugu programa idatziko. Zehatzago esanda, hertsiki inplementatutako moten berdintasuna programatzea ez da beharrezkoa izango.

4.3. Inplementazioen erabilera

Behin **T** datu-motaren inplementazioa eraikita, gure makroprogrametan erabiltzearen helburua bete dezakegu, baina nola eta zergatik zehaztea komeni da.

Lehendabizi, motako konstanteak makroadierazpenetan eta makrobaldintzetan erabili ahal izango ditugu. $\mathbf{t} \in \mathbf{T}$ konstantearen agerpenak dituen edozein makroren hedapenaren lehen pausoan agerpen horiek *'w'*rekin ordezkatuko dira, $w = \Re_{\mathbf{T}}(\mathbf{t})$ izanik (**t**-ren adierazpide kanonikoa). STRING

motako konstanteekin egiten genuen bezala, KONPUTAGARRIAK paketean dagokion funtzio konstantea ez dugu erazagutu beharko.

Hedapenak errealitate gorrira gakartza: **T** motako objektuak erabiltzen genituelako irudipena genuen, baina gure interpretazioak "mozorroturiko" hitzak dira soilik. Dena dela, horrek ez gaitu arduratuko: ez zaizkigu **T** motaren osagaiak eta eragiketak interesatzen *bere osagaien eta eragiketen ezaugarriak* baizik. Adibidez, zenbakizko konputazioa egin nahi dugunean ez da zenbakiekin lan egitea bereziki gogoko dugulako, zenbatzea, batzea, biderkatzea, etab. interesatzen zaigulako baizik, eta hori guztia gure while programekin egitea edukiko dugu.

Bestalde, **T** motako eragiketak makroadierazpenetan eta makrobaldintzetan erabili ahal izango ditugu. ψ eragiketaren agerpenak dituen makroprograma badugu, bere hedapenaren lehen pausoa agerpen horiek dagokion hitzen arteko funtzioa den φ_P^j -rekin ordezkatzea izango da (non P ψ inplementatzen duen programa den). Funtzio hau aipatzea ere ez da beharko, datu-mota erazagutzen dugunean bere inplemantazioan sartzen diren oinarrizko funtzio guztiak prezio berean sartzen direla suposatuko baitugu.

Mota BOOLEARRA definitu dugunetik, **T** motaren osagaiak makroprogrametan erabiltzeko hirugarren modu bat dago: bere predikatuak (hots, emaitza BOOLEARRA sortzen duten eragiketa osoak) makrobaldintzetan erabil daitezke. Hedapena aurrekoarenaren parekoa da.

Azkenik, kateekin lan egiten duten zenbait funtzio while-konputagarri edozein datu-motak arazorik gabe hereda ditzake: identitatea, proiekzioak, funtzio hutsa edo funtzio konstanteak. Funtzio hauekin arazotxoa sortuko litzaiguke gure makroprogramen goiburukoetan erazagutuko bagenitu, dexente handituko bailituzketelako. Zorionez, kateen arteko funtzioetarako geneukan hitzarmenari jarraituko diogu, eta ez ditugu sartu beharko.

Dudarik gabe, inplementatutako datu-mota gure programak eraikitzeko erabiltzen hastean, ondorio zuzenetariko bat mota horri lotutako funtzio berriak eraikitzea izango da. Inplementazioaren ostean definitutako funtzio horiek motaren **aberasketak** direla esango dugu. Puntu honetan while-konputagarritasunaren kontzeptua zabaltzea komeni da.

18. DEFINIZIOA: Bedi edozein $\Psi: T_1 \times \ldots \times T_j \rightarrow T_{j+1}$ funtzio, $T_1, T_2, \ldots, T_{j+1}$ mota guztiak inplementatuta izanik. Ψ while-konputagarria dela esango dugu funtzioa inplementatzen duen **P** while programa existitzen denean. Ψ osoa bada eta $T_{j+1} = \mathbf{B}$ betetzen bada, orduan predikatua izango da eta while-errekurtsiboa

dela ere esan dezakegu. Ψ while-konputagarria dela frogatzean, parte hartzen duten mota guztiek, STRING ezik, bere goiburukoan agertu behar dute.

Garrantzi gutxiko desberdintasun bat eta garrantzi handiko beste bat daude mota sortzerakoan definitzen diren eta aberasketa diren funtzioen artean. Lehenengoak eraikitzean, datu-mota oraindik inplementaturik ez dagoenez, *zuzenean adierazpidearen gainean lan egin behar da*, hitzen gaineko eragiketak erabiliz. Datu-mota inplementatu ondoren, ordea, aldagaia benetan mota honetakoa bailitz lan egiten uzten digun eragiketa-multzoa izango dugu. Horregatik, aberasketetan *inplementazioa diseinatu den modu zehatzarekiko menpekotasuna duen edozer erabiltzea ekidingo dugu*, motako konstanteak eta eragiketak erabiliko ditugu soilik. Adibidez, edukin boolearra duen aldagai bat erabiltzean, sekula ere ez dugu *nonem?* predikatua bere gainean erabiliko, izatekotan bere edukina *true* den galdetuko dugu. Honen atzean dagoen ideia inplementazioaren **ikuspen eza** da, bai datuen adierazpideari bai eragiketen inplementazioari dagokionean.

17. ADIBIDEA : Ondoko funtzio eta predikatuak while-konputagarriak, eta hortaz, BOOLEARRA eta ARRUNTA moten aberasketak direla frogatuko dugu.

a) Konjuntzio eta disjuntzio funtzioak, mota BOOLEARRAri ohiko zentzuaz aplikatzen zaizkionak:

$$\wedge: \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$$

$$\vee: \mathbf{B} \times \mathbf{B} \rightarrow \mathbf{B}$$

Konjuntzioa mota eraikitzean definitutako eragiketetan oinarrituta konputa dezakegu (inplementazioko funtzioak erazagutzeko beharrik ez dagoela ohar ezazu):

```
package KONPUTAGARRIAK is
    type BOOLEARRA is STRING;
end KONPUTAGARRIAK;

if (X1 <-> true and X2 <-> true) then X0 := true; else X0 := false; end if;
```

eta disjuntzioa eragiketa baita ere, definitu berrian oinarritzen dugula suposatuz:

```
package KONPUTAGARRIAK is
    type BOOLEARRA is STRING;
    function "∧" (BOOLEARRA, BOOLEARRA) return BOOLEARRA;
end KONPUTAGARRIAK;

X0 := ¬(¬X1 ∧ ¬X2);
```

b) Zenbaki arrunten artean definitzen den aurrekaria funtzioa, eta arrunten arteko konparaketa-predikatuak konputagarriak direla frogatzea ez da zaila:

$$\text{pred}: \mathbf{N} \rightarrow \mathbf{N}$$

$$<: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$<=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$>: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$>=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

$$/=: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{B}$$

Ez dugu eragiketa hauei dagozkien programak egin beharrik izango, guztiak ere kateen artean definitutakoei baitagozkie. *pred*-en kasuan *aurre* funtzioa da eta gainerakoetan izen bera duten hitzen arteko predikatuak.

c) Batuketa, kenketa eta biderkaketa funtzio aritmetikoak:

$$+: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

$$-: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

$$*: \mathbf{N} \times \mathbf{N} \rightarrow \mathbf{N}$$

suc funtzioa behin eta berriz aplikatuz batuketa egitea erraza da:

```
package KONPUTAGARRIAK is
      type ARRUNTA is STRING;
      type BOOLEARRA is STRING;
      function "/=" (ARRUNTA, ARRUNTA) return BOOLEARRA;
      function pred (ARRUNTA) return ARRUNTA;
end KONPUTAGARRIAK;
```

```
X0 := X1;
LAG := X2;
-- X0+LAG=X1+X2
while LAG /= 0 loop
      X0 := suc(X0);
      LAG := pred(LAG);
end loop;
```

Kenketarako aurreko programaren goiburuko bera erabiliko dugu. Zenbaki arrunten arteko kenketa problematikoa dela kontuan izan behar dugu, izan ere ezin dugu emaitza negatiborik adierazi. Gure kasuan, kenkizuna kentzailea baino txikiagoa denean emaitza 0 bezala definituko dugu (*pred* funtzioaren definizioarekin eta erroreak ekiditeko gure ohiturarekin bat etorriaz):

```
   X0 := X1;
   LAG := X2;
-- X0–LAG=X1–X2
   while LAG /= 0 loop
         X0 := pred(X0);
         LAG := pred(LAG);
   end loop;
```

Biderkaketarako goiburuko berria egingo dugu, batuketa erabili nahiean:

```
   package KONPUTAGARRIAK is
         type ARRUNTA is STRING;
         type BOOLEARRA is STRING;
         function "/=" (ARRUNTA, ARRUNTA) return BOOLEARRA;
         function "+" (ARRUNTA, ARRUNTA) return ARRUNTA;
         function pred (ARRUNTA) return ARRUNTA;
   end KONPUTAGARRIAK;

   X0 := 0; LAG := X2;
-- X0+X1*LAG=X1*X2
   while LAG /= 0 loop
         X0 := X0 + X1;
         LAG := pred(LAG);
   end loop;
```

Gainera, zenbaki arruntak eta hitzak argumentu dituzten funtzioen while-konputagarritasuna ere froga dezakegu. Oro har funtzio hauek STRING motaren aberasketa mistoak izango dira.

Adibidez, *luzera* funtzioa, hitz bat emanik bere ikur-kopurua itzultzen duena, while-konputagarria dela ikus daiteke, bere definizio induktiboaz:

- $|\varepsilon| = 0$
- $|x \bullet \mathbf{s}| = \mathrm{suc}(|x|)$

Eta inplementatuko duen programa ondokoa izan daiteke:

```
   package KONPUTAGARRIAK is
         type ARRUNTA is STRING;
   end KONPUTAGARRIAK;

   X0 := 0;
   LAG := X1;
   while nonem?(LAG) loop
         X0 := suc(X0);
         LAG := cdr(LAG);
   end loop;
```

Badakigu edozein hitz *k*-tupla baten kodetzat har dezakegula. *k* zenbakia ezaguturik, *k*-tupla horren *j*. osagaia ematen duen $deskod_{k,j}$ edozein deskodeketa-funtzio konputa daiteke. Baina beste zerbait orokorragoa ere egin dezakegu: funtzio-multzo bat definitu beharrean, guztiak biltzen dituen funtzio bakarra defini dezakegu:

$$\text{deskod}: \mathbf{N} \times \mathbf{N} \times \Sigma^* \rightarrow \Sigma^*$$

non edozein *w* izanik deskod(k, j, w) = $\text{deskod}_{k,j}(w)$ betetzen den eta *k=0*, *j=0* edo *j>k*-rako indefinitua den.

Funtzio honen abantaila hau da: programa idazterakoan ez dugu zertan jakin behar ez *w*-k zenbat hitz kodetzen dituen ezta horietako zein interesatzen zaigun, eta honela datu hauek egikaritzapenean zehaztuko dira. Gainera *deskod*-en konputagarritasuna ez dator $deskod_{k,j}$ funtzio guztien while-konputagarritasunetik (berez azken hauek kasik ez dira erabiltzen programan). Kapitulu honen hasieran aipatu orokortzearen adibide bat da.

```
package KONPUTAGARRIAK is
      type ARRUNTA is STRING;
      type BOOLEARRA is STRING;
      function "<" (ARRUNTA, ARRUNTA) return BOOLEARRA;
      function pred (ARRUNTA) return ARRUNTA;
      function deskod_2_1 (STRING) return STRING;
      function deskod_2_2 (STRING) return STRING;
end KONPUTAGARRIAK;

if (X1 = 0) or (X2 = 0) or (X1 < X2) then
      X0 := ⊥⊥ ;
else  LAG := X3;
      if X1 = X2 then
            MAX := pred(X2);
      else   MAX := X2;
      end if;
      for IND in 1 .. MAX loop
            X0 := deskod_2_1(LAG);
            LAG := deskod_2_2(LAG);
      end loop;
      if X1 = X2 then
            X0 := LAG;
      end if;
end if;
```

4.4. Mota konposatuen inplementazioa

Deskribatutako mekanismoak **mota konposatuak** inplementatzeko baliagarriak direla ikusiko dugu orain. Mota konposatu hauek beste mota bateko (**oinarri-mota** deitutakoa) balioen elkarketaren bidez osatuko dira. Ikusiko ditugun bi adibideek (pilek eta bektore dinamikoek) Konputagarritasun Teoriaren garapenean oinarrizko algoritmoak eraikitzerakoan erakutsiko dute euren erabilgarritasuna.

Bi kasuetan STRING erabiliko dugu oinarri-mota gisa, baina, esate baterako, zenbaki arrunten pilen bektoreak eraiki nahi izanez gero, mota konposatu hauek zailtasun handirik gabe birdefini ditzakegu. Horretarako, dagozkien moten gaineko eragiketak erazagutu eta oinarri-motaren osagaiak modu egokian interpretatu besterik ez da egin beharko.

4.4.1. PILA mota

Pilak datuak gordetzeko egiturak dira. PILA datu-mota hitzen pilen **P** multzoak osatzen du. Motako balioak zero karaktere-katea edo gehiagoren elkarketaren bidez osatzen dira; katea horiek, pilaren **osagaiak** deitutakoak, ordenan gordetzen dira. Pila motako objektuak bere osagaiak "<" eta "]" ikur artean listatuta adieraziko ditugu. Adibidez, < **aba**, **bb**, ε, **a**] lau osagai dituen pila bat da eta <] zero osagai dituen pila bat, **pila hutsa** deitutakoa. Pila ez-huts batean "<" ikurretik gertuen dagoen osagaiari pilaren **tontorra** deituko diogu. PILA motaren oinarrizko eragiketak honakoak dira:

$$\text{pilaratu: } \Sigma^* \times \mathbf{P} \rightarrow \mathbf{P}$$

$$\text{pila\_hutsa\_da?: } \mathbf{P} \rightarrow \mathbf{B}$$

$$\text{tontorra: } \mathbf{P} \rightarrow \Sigma^*$$

$$\text{despilatu: } \mathbf{P} \rightarrow \mathbf{P}$$

Interpretazio-funtzioa definitzeko pila bakoitza hitz batez adierazi behar dugu, osagai guztiak sartu direneko ordena errespetatuz kodetu behar dituena. Eta, jakina, informazio horrek berreskuragarria izan behar du, beraz, **3.6**. atalean definitutako kodeketa-funtzioa erabiltzeak zentzuzkoa ematen du. $< x_1, x_2, ..., x_k]$ pila $kod^k(x_1, x_2, ..., x_k)$ bezala kodetzeak errazena dirudi, baina ez litzateke oso ideia ona, interpretazio-funtziorik ziurtatuko ez ligukeelako:

- Batetik, <] pila hutsak ez du kodetu beharreko osagairik, eta, hala ere, kodetu behar dugu. Horretarako kode bat gorde beharko dugu.

- Bestetik, demagun, adibidez, alfabetoa {**a**, **b**, **c**} dela, nola interpretatu behar da **acba** hitza, **acba** = kod^1(**acba**) = kod^2(**aa**, **ac**) = kod^3(**aa**, **c**, ε) = kod^4(**aa**, **c**, ε, ε) = ... izanik? Bere kodetik abiatuta pilak zenbat osagai dituen jakiteko moduren bat ezarri behar dugu.

Azken arazo hau ebazteko modu bat hauxe da, pentsa dezagun pilatik ε topatu arte osagaiak "atera" ditzakegula, hitz hutsak "pilaren hondoa"ren funtzioa betetzen duelarik. Honela, $deskod_{2,2}$ behin eta berriz aplikatuz derrigorrez ε lortzearen abantaila dugu (ez da zaila $deskod_{2,2}(x) < x$ dela frogatzea). Sistema honekin lehenengo arazoa ere konpontzen dugu: ε-ek ateratzeko ezer ez duen pila adieraziko luke. Baina orain beste zailtasun bat dugu, izan ere, honela ezin ditugu hondoan hitz hutsa duten pilak adierazi (hitz hori pila hutsik dagoelako seinalearekin nahasiko genuke).

Benetako ebazpidea truko txiki batez lortuko dugu: pilako osagai guztiak normalki kodetuko ditugu *hondokoa ezik*, hitz honen ordez bere hurrengoa kodetuz. Hau da, $\Re_{\mathbf{P}}(< x_1, x_2, ..., x_k]) = kod^{k+1}(x_1, x_2, ..., hur(x_k), \varepsilon)$ izatea nahi dugu. Honela, hitz bat interpretatzean eta kodetzen dituen hitzak lortzen ditugunean, ziur gaude bukaerako ε-ek pilaren bukaera adierazten duela.

Dena dela, garrantzitsuena interpretazio-funtzioa da, izan ere hori gabe ez baitugu inplementaziorik. Aipatu ditugun ideiak jarraituz:

$$\Im_{\mathbf{P}}(w) =$$

$$\begin{cases} <] & \text{baldin } w=\varepsilon \\ < aurre(deskod_{2,1}(w))] & \text{baldin } w\neq\varepsilon \wedge deskod_{2,2}(w)=\varepsilon \\ pilaratu\ (deskod_{2,1}(w);\ \Im_{\mathbf{P}}(deskod_{2,2}(w))) & \text{bestela} \end{cases}$$

Funtzioa osoa da lehen komentatu dugunarengatik: beti $deskod_{2,2}(x) < x$ betetzen denez, definizio errekurtsiboa edozein *w*-rako konbergentea delakoaren garantia dugu. Bijektiboa dela ere erraz ikus daiteke. Orain bere eragiketak inplementatzen dituzten programak definituko ditugu. Guztientzat beharko dugun goiburuko amankomuna honako hau izango da:

```
package KONPUTAGARRIAK is
    function kod_2 (STRING, STRING) return STRING;
    function hur (STRING) return STRING;
    type BOOLEARRA is STRING;
    function deskod_2_2 (STRING) return STRING;
    function deskod_2_1 (STRING) return STRING;
    function aurre (STRING) return STRING;
end KONPUTAGARRIAK;
```

Lehenik pilaren tontorrean hitza gehitzen duen *pilaratu* funtzioa dugu:

if nonem?(X2) **then** X0 := kod_2(X1, X2);
else X0 := kod_2(hur(X1), ε); **end if**;

pila_hutsa_da? predikatua honako programa honek konputatzen du:

if nonem?(X1) **then** X0 := false; **else** X0 := true; **end if**;

Pila ez-hutsa emanda, bere tontorrean dagoen hitza itzultzen duen *tontorra* funtzioa ere inplementatu behar dugu. Aitzitik, pila hutsa bada, definitu gabe dago.

```
if  nonem?(X1) then
      if nonem?(deskod_2_2 (X1)) then
            X0 := deskod_2_1 (X1);
      else   X0 := aurre(deskod_2_1 (X1));
      end if;
else  X0 := ⊥⊥ ;
end if;
```

Azkenik, argumentuan pasa zaion pilaren tontorrean dagoen osagaia ezabatu ondoren geratzen den pila itzultzen duen *despilatu* funtzioa inplementatzeko:

if nonem?(X1) **then** X0 := deskod_2_2 (X1);
else X0 := ⊥⊥ ;
end if;

4.4.2. BEKTORE mota

Bektore dinamikoak ere egikaritzapen-denboran tamaina alda dezaketen eta datuak ordenan gordetzen dituzten egiturak dira. Pilarekiko diferentzia hauxe da, bektore baten edozein osagai edozein unetan bere indizearen bitartez zuzenean atzi daitekeela. Indizeak zerotik hasita zenbatzen dira beti eta edozein bektorek gutxienez osagai bat du. Datu-motaren balioen multzoari **V** deituko diogu.

Bektoreak parentesien arteko osagai zerrenden bidez adieraziko ditugu. Honela, (**aa**, **b**, **aca**) bektorearen 0. osagaia **aa** eta bigarren osagaia **aca** izango dira. Bektorea tamaina zehatzarekin sortzen da, eta bere edozein osagai alda daiteke. Indizea handiegia delako existitzen ez den osagai bat aldatzen saiatzen den momentuan, bektoreak bere tamaina aldatuko du osagai berria gorde ahal izateko, aldi berean tarteko indize guztien posizioetan hitz hutsa sartuko delarik. Aurreko adibidean bektorearen zazpigarren osagaia aldatzearekin batera bektorearen

tamaina zortzira pasako da, hirugarrenetik seigarrenerako osagaiak ε bihurtuz. Aldiz, ezin da existitzen ez den osagaia atzitu.

Datu-mota honetarako hurrengo eragiketak definituko ditugu:

$$\text{azken\_indizea: } \mathbf{V} \rightarrow \mathbf{N}$$

$$\text{edukina: } \mathbf{V} \times \mathbf{N} \rightarrow \Sigma^*$$

$$\text{aldatu: } \mathbf{V} \times \mathbf{N} \times \Sigma^* \rightarrow \mathbf{V}$$

Interpretazio-funtzioa definitzeko pilekin genuen arazo berbera dugu: $(x_0, x_2, ..., x_n)$ bektorea $kod^{k+1}(x_0, x_1, ..., x_k)$ hitzaren bidez adieraztea ez da egokia, osagaiak ezin ditugulako berreskuratu. Baina kasu honetan bektorearen tamaina kodeketan bertan gordeaz ebatziko dugu arazoa, hau da, aipatutako bektorea $kod^{k+2}(k, x_0, x_1, x_2, \ldots, x_k)$ adieraziz. Jakina, funtzioaren lehen argumentua k zenbakia adierazten duen hitza da. Beraz, erabiliko dugun interpretazio-funtzioa hauxe izango da:

$$\Im_{\mathbf{V}}(w) = (x_0, x_1, x_2, \ldots, x_k) \Leftrightarrow$$
$$\begin{cases} k = \Im_{\mathbf{N}}(deskod_{2,1}(w)) \\ \wedge \\ \forall i\ (0 \le i \le k \rightarrow z_i = deskod_{k+1,i+1}(deskod_{2,2}(w))) \end{cases}$$

Inplementatu beharrekoetatik, *azken_indizea* funtzioa, bektorearen azken osagaiari dagokion indize edo posizioa ematen diguna, berez $deskod_{2,1}$ da, eta ez dugu programatu behar izango. *edukina* funtzioak bektore bat eta zenbaki bat (atzitu nahi dugun posizioa) emanda, posizio horretan dagoen osagaia ematen digu. Lehen esan dugun bezala, funtzio hau bektorearen azken posizioa baino handiagoak diren indizeetarako definitu gabe dago.

```
package KONPUTAGARRIAK is
    type ARRUNTA is STRING;
    function "<=" (ARRUNTA, ARRUNTA) return BOOLEARRA;
    function deskod (ARRUNTA, ARRUNTA, STRING) return STRING;
    function deskod_2_1 (STRING) return STRING;
    function deskod_2_2 (STRING) return STRING;
end KONPUTAGARRIAK;
```

Programak funtzioa definitu gabe dagoeneko kasua bereizten du, eta beste kasuetan *deskod* funtzioa erabiltzen du eskatutako osagaia lortzeko:

```
if X2 <= deskod_2_1(X1) then
    X0 := deskod(suc(deskod_2_1(X1)), suc(X2), deskod_2_2(X1));
```

```
else X0 := ⊥;
end if;
```

Azkenik, *aldatu* funtzioa dugu, bektore bat, posizio bat eta hitz bat hartuta hitza emandako posizioan sartu ondoren lortutako bektorea itzultzen duena. Printzipioz,*aldatu* funtzioak emandako posizioan dagoen hitza ordezkatzen du. Baina aldatu beharreko posizioa bektorearen azkena baino handiagoa bada, posizio hori eta aurreko guztiak automatikoki sortuko ditu, azken hauek ε balioaz beteko dituelarik. Dudarik gabe hauxe da funtzioetan inplementatzen zailena, izan ere, edozein osagai aldatzeko bektoreari literalki barrenak atera eta bere egitura goitik behera berregin beharko dugu, aurreko antolaketa ezertarako aprobetxatzeko aukerarik gabe.

```
package KONPUTAGARRIAK is
    type PILA is STRING;
    type ARRUNTA is STRING;
    type BOOLEARRA is STRING;
    function deskod_2_1 (STRING) return STRING;
    function deskod_2_2 (STRING) return STRING;
    function pred (ARRUNTA) return ARRUNTA;
    function "<" (ARRUNTA, ARRUNTA) return BOOLEARRA;
    function kod_2 (STRING, STRING) return STRING;
    function "=" (ARRUNTA, ARRUNTA) return BOOLEARRA;
end KONPUTAGARRIAK;
```

aldatu funtzioak hiru kasu bereiziko ditu: posizioa azkena baino txikiagoa izatea, azkenekoa baino handiagoa izatea ala zehazki azkena izatea. Edozein kasutan ere bektorea "desegitean" bere osagaiak pila batean gordeko ditugu.

```
- - AZK_IND aldagaiak bektorearen azken indizea gordeko du, eta LAGek bere "mamia"
   AZK_IND := deskod_2_1(X1);
   LAG := deskod_2_2(X1);
- - Lehendabizi PILALAG pilan gordeko ditugu bektorearen osagaiak
   PILALAG := < ];
- - Iterazio bakoitzean bektorearen i. osagaia gehitzen zaio pilari
   for IND in 0 .. pred(AZK_IND) loop
        PILALAG := pilaratu(deskod_2_1(LAG), PILALAG);
        LAG := deskod_2_2(LAG);
   end loop;
- - Tontorrean azken osagaia egongo da
   PILALAG := pilaratu(LAG, PILALAG);
   if X2 < deskod_2_1(X1) then
- - 1. kasua: azkenekoaren aurretik dagoen posizio bat  aldatu nahi da
        X0 := tontorra(PILALAG);
```

```
-- Emaitzan aldatu beharrekoaren ondoren dauden osagaiak kargatzen ditugu
        for IND in reverse pred(AZK_IND) .. suc(X2) loop
            PILALAG := despilatu(PILALAG);
            X0 := kod_2(tontorra(PILALAG), X0);
        end loop;
-- Balio berria kargatu eta zaharra pilatik kenduko dugu
-- (aurreko begiztan kendu gabe geratu den bestearekin batera)
        X0 := kod_2(X3, X0);
        PILALAG := despilatu(despilatu(PILALAG));
    else X0 := X3;
-- 2. kasua:bektorearen azken posizioa edo ondorengo bat aldatu nahi da
-- Edozein kasutan balio berria emaitzaren azken osagaia izango da
-- Posizio berriak sortzen direnean hitz hutsarekin beteko dira
        for IND in reverse pred(X2) .. suc(AZK_IND) loop
            X0 := kod_2(ε, X0);
        end loop;
-- Azken posizioa aldatzen bada, bere balio zaharra pilatik kenduko da
        if X2 = AZK_IND then PILALAG := despilatu(PILALAG); end if;
-- Edozein kasutan aldatutako posizioak indize maximo berria izango du
        AZK_IND := X2;
    end if;
-- Hemendik aurrera kasu guztiak biltzen dira: pilatik aldatutako posizioaren aurreko
-- balioak gorde eta emaitza bektorea eraikitzen bukatzea besterik ez da falta
    while not pila_hutsa_da?(PILALAG) loop
        X0 := kod_2(tontorra(PILALAG), X0);
        PILALAG := despilatu(PILALAG);
    end loop;
    X0 := kod_2(AZK_IND, X0);
```

OHARRA: ADA-n bezala, **A(α)** makroadierazpena erabiliko dugu **edukina(A,α)**ren ordez, non **A** bektore motako aldagaia eta α adierazpen arrunta diren. Halaber, bektore baten gainean *aldatu* funtzioa erabiltzen den kasu gehienetan emaitza bektore berari esleituko diogu, esleipena **A:=aldatu(A,α,β);** izanik, non β edozein makroadierazpen den. Kasu hauetan ere **A(α):=β;** erako notazioa onartuko dugu, **bektoreen osagaien gaineko makroesleipenak** deituko ditugularik.

4.5. While programen Gödelizazioa

Gure while programetan maneiatu nahi izango ditugun azken objektuak WHILE mota osatuko duten while programak berak izango dira. Datu-mota honen balio-multzoari **W** deituko diogu. Inplementatzen ditugun unetik, programazioarekin zerikusia duten funtzio askoren while-konputagarritasuna aztertu ahal izango dugu. Eragiketa hauetako batzuk ezaugarri estatikoei

dagozkie programen testutik ondoriozta daitezkeelako (konpilazio-denboran ebaluatuko dira), eta beste batzuk, berriz, ezaugarri dinamikoei, programen funtzionamenduaren menpe daudelako (egikaritzapen-denboran ebaluatuko dira).

Informatikan programa batek beste programa batzuen gainean lan egitea gauza arrunta den arren, ikuspegi teorikotik izugarrizko garrantzia du. Berez, sistema formal baten osagaiak sisteman bertan maneiatzean datzan murgiltze mekanismo honek lehenengo aldiz deskribatu zuen pertsonaren izena hartu du: **gödelizazioa**.

Ohi bezala, inplementazio hertsia lortu nahi dugu. Erabiliko dugun ideia makina-lengoaia askotan erabili denaren antzekoa da: programa bakoitza *(eragiketa_kodea, informazioa)* bikote baten bidez kodetuko da, non *eragiketa_kodea*k zein programa mota den deskribatuko du, eta *informazioa*ren bidez adierazgarriak diren beste gehigarriak aipatuko dira (eragiketaren parametroak eta argumentuak, batez ere). Beraz, dauden sei while programa motetarako sei eragiketa-kode beharko dugu. Baina interpretazio-funtzioak osoa izan behar du, eta sei kode hauek besterik ez baditugu, nola interpretatuko ditugu sei hauetan ez dagoen eragiketa-kodea duten hitzak? Horrelako hitzak dagoeneko kodea duten beste programei ausaz berregokitzea erabaki genezake, baina orduan ez genuke injektibotasunik izango.

Ebazpidea mota zehatz bateko programen artean kode-kopuru infinitua banatzean datza. While programak *n* ikur dituen alfabeto baten gainean definiturik badaude, eragiketa-kodeen erreserba hau egin dezakegu:

- **0** kodea **XI:=Ɛ;** erako while programetarako
- **1** ≤ k ≤ **n** kodeak **XI:=cons$_{a_k}$(XJ);** erakoetarako
- **n+1** kodea **XI:=cdr(XJ);** erakoetarako
- **n+2** kodea **P$_1$ P$_2$** konposaketetarako
- **n+3** ≤ k ≤ **2*n+2** kodeak **<u>if</u> car$_{a_k}$?(XI) <u>then</u> P <u>end</u> <u>if</u>;** baldintzetarako
- balio handiagoak **<u>while</u> nonem?(XI) <u>loop</u> P <u>end</u> <u>loop</u>;** begiztetarako.

Eragiketa-kodearen barruan bere klasea ez ezik programari buruzko informazio gehiago ere jartzeak bijektibotasuna lortzearren da. Adibidez, *cons* funtzioan edo *car* predikatuan parte hartzen duen ikurraren mota ezaugarri independente gisa kodetuko bagenu, honako arazo hau izango genuke: *1* eta *k* tarteko balioak soilik hartu ahal izango ditu, beraz, osagai horretan beste balio bat

duten hitzek printzipioz ez lukete inongo programarik errepresentatuko (bere adierazpide propioa duen programaren bat bezala interpretatu beharko lirateke ausaz). **while**-en kasua are argiagoa da: programa hauen artean *2*n+2* baino handiagoak diren eragiketa-kode guztiak banatuko dira, kodeak kasu bakoitzean programari buruzko informazio berezia izango duelako.

Aukeratutako interpretazioa hobeto ulertzeko, sortu nahi dugun adierazpide-funtzioa aurkeztea komeni da, honela errazago ikusiko dugu hitzen bidez kodetu nahi dugun "informazio" hori zein den (gogoan izan *n*-k $\Sigma = \{\mathbf{a_1}, \mathbf{a_2}, \ldots, \mathbf{a_n}\}$ alfabetoaren kardinala adierazten duela):

$$\Re_{\mathbf{W}}(\mathbf{XI:=\varepsilon;}) = \text{kod}^2(\mathbf{0}, \text{i})$$

$$\Re_{\mathbf{W}}(\mathbf{XI:=cons_{a_k}(XJ);}) = \text{kod}^2(\mathbf{k}, \text{kod}^2(\text{i}, \text{j})) = \text{kod}^3(\mathbf{k}, \text{i}, \text{j})$$

$$\Re_{\mathbf{W}}(\mathbf{XI:=cdr(XJ);}) = \text{kod}^2(\mathbf{n+1}, \text{kod}^2(\text{i}, \text{j})) = \text{kod}^3(\mathbf{n+1}, \text{i}, \text{j})$$

$$\Re_{\mathbf{W}}(\mathbf{P_1\ P_2}) = \text{kod}^2(\mathbf{n+2}, \text{kod}^2(\Re_{\mathbf{W}}(\mathbf{P_1}), \Re_{\mathbf{W}}(\mathbf{P_2})))$$

$$\Re_{\mathbf{W}}(\underline{\mathbf{if}}\ \mathbf{car_{a_k}?(XI)}\ \underline{\mathbf{then}}\ \mathbf{P}\ \underline{\mathbf{end}}\ \underline{\mathbf{if}}\mathbf{;}) = \text{kod}^2(\mathbf{n+2+k}, \text{kod}^2(\text{i}, \Re_{\mathbf{W}}(\mathbf{P})))$$

$$\Re_{\mathbf{W}}(\underline{\mathbf{while}}\ \mathbf{nonem?(XI)}\ \underline{\mathbf{loop}}\ \mathbf{P}\ \underline{\mathbf{end}}\ \underline{\mathbf{loop}}\mathbf{;}) = \text{kod}^2(\mathbf{2*n+3+i}, \Re_{\mathbf{W}}(\mathbf{P}))$$

Honela kodetu nahi ditugu programaren gainerako osagaiei buruzko datuak: esleipen hutsaren kasurako *i* aldagaiaren balioa behar dugu, *cons* eta *cdr* motako esleipenetarako parte hartzen duten *i* eta *j* aldagaien indizeak behar ditugu, etab. Laburbilduz, interpretazio-funtzioa hauxe izango litzateke:

$$\Im_W(w) = \begin{cases} \text{XI:} = \varepsilon; & \text{baldin } \Im_N(\text{deskod}_{2,1}(w)) = 0 \wedge \\ & \wedge i = \Im_N(\text{deskod}_{2,2}(w)) \\ \text{XI:} = \text{cons}_s(\text{XJ}); & \text{baldin } 1 \le \Im_N(\text{deskod}_{2,1}(w)) \le n \wedge \\ & \wedge i = \Im_N(\text{deskod}_{3,2}(w)) \wedge \\ & \wedge s = \text{deskod}_{3,1}(w) \wedge \\ & \wedge j = \Im_N(\text{deskod}_{3,3}(w)) \\ \text{XI:} = \text{cdr}(\text{XJ}); & \text{baldin } \Im_N(\text{deskod}_{2,1}(w)) = n+1 \wedge \\ & \wedge \Im_N(\text{deskod}_{3,2}(w)) = i \wedge \\ & \Im_N(\text{deskod}_{3,3}(w)) = j \\ \text{P Q} & \text{baldin } \Im_N(\text{deskod}_{2,1}(w)) = n+2 \wedge \\ & \wedge \Im_W(\text{deskod}_{3,2}(w)) = P \wedge \\ & \wedge \Im_W(\text{deskod}_{3,3}(w)) = Q \\ \text{if } \text{car}_s?(\text{XI}) \text{ then P end if}; & \text{baldin } n+3 \le \Im_N(\text{deskod}_{2,1}(w)) \le 2*n+2 \wedge \\ & \wedge \Im_N(\text{deskod}_{3,2}(w)) = i \wedge \\ & \wedge \Im_N(\text{deskod}_{3,1}(w)) - n - 2 = \Im_N(s) \wedge \\ & \wedge \Im_W(\text{deskod}_{3,3}(w)) = P \\ \text{while nonem?(XI) loop P end loop}; & \text{baldin } \Im_N(\text{deskod}_{2,1}(w)) \ge 2*n+3 \wedge \\ & \wedge i = \Im_N(\text{deskod}_{2,1}(w)) - 2*n - 3 \wedge \\ & \wedge \Im_W(\text{deskod}_{2,2}(w)) = P \end{cases}$$

Inplementatu beharreko eragiketez hiru multzo egin dezakegu: lehendabizi funtzio eraikitzaileak ditugu. Hauei esker WHILE motako objektuak induktiboki osatuko ditugu, euren eraikuntzan parte hartzen duten aldagai, ikur edota azpiprogramen datuetatik abiatuta. Orduan ondorengo programa motetarako eragiketa eraikitzaile bana izango dugu:

$$\text{esleipen\_hutsa\_eratu: } \mathbf{N} \to \mathbf{W}$$

$$\text{esleipen\_cons\_eratu: } \mathbf{N} \times \Sigma^* \times \mathbf{N} \to \mathbf{W}$$

$$\text{esleipen\_cdr\_eratu: } \mathbf{N} \times \mathbf{N} \to \mathbf{W}$$

$$\text{konposaketa\_eratu: } \mathbf{W} \times \mathbf{W} \to \mathbf{W}$$

$$\text{baldintzazkoa\_eratu: } \Sigma^* \times \mathbf{N} \times \mathbf{W} \to \mathbf{W}$$

$$\text{iterazioa\_eratu: } \mathbf{N} \times \mathbf{W} \to \mathbf{W}$$

Funtzio guztietarako argumentu arruntek eraikuntzan parte hartzen duten aldagaien indizeak adieraziko dituzte. Horretaz gain, Σ datu-mota definiturik ez daukagunez (finitua izanik inplementazio ez-hertsia egitera behartuko gintuzke),

esleipen_cons_eratu eta *baldintzazkoa_eratu* funtzioak definitzerakoan *cons* funtzioan zein *car* predikatuan ageri behar den **s** ikurra adierazteko STRING motako argumentua sartu dugu, funtzio eraikitzaileek bere lehenengo ikurra baino ez dutela kontutan hartuko suposatuz (eta hitz hutsa bada dibergituko dutela, noski).

Bestalde, ikuskatze predikatuak ere oso erabilgarriak izango dira. Predikatu hauek programa bat emanik mota zehatz batekoa denentz emaitza boolearraren bidez itzuliko digute. Programa-mota bakoitzeko predikatu bat izango dugu.

esleipen_hutsa_da?: **W → B**

esleipen_cons_da?: **W → B**

esleipen_cdr_da?: **W → B**

konposaketa_da?: **W → B**

baldintzazkoa_da?: **W → B**

iterazioa_da?: **W → B**

Azkenik, while programan parte hartzen duten osagaiak emango dizkiguten atzipen-eragiketak ere inplementatuko ditugu; while programaren motaren arabera jakin dezakegu, esate baterako, zein aldagairi esleitzen zaion balioa, edota zein aldagaik kontrolatzen duen baldintza, edota zein azpiprogramak parte hartzen duen eraikuntzan, etab. Adibidez:

aldagai_indize: **W → N**

funtzioak, programak erabiltzen duen aldagai nagusiaren indizea adierazten du (hau da, lehen aipatzen denarena eta, beraz, kabitze-mailarik altueneko aginduan agertzen denarena). Programaren motaren arabera esanahi anitz izan dezake:

- aldagai_indize(**XI:=ε;**) = i
- aldagai_indize(**XI:=cons$_{ak}$(XJ);**) = i
- aldagai_indize(**XI:=cdr(XJ);**) = i
- aldagai_indize(**if car$_{ak}$?(XI) then P end if;**) = i
- aldagai_indize(**while nonem?(XI) loop P end loop;**) = i

Programa konposatuetan *aldagai_indize* funtzioa definitzeak ez dauka zentzu handirik, programa eraikitzerakoan inolako paperik jokatzen ez duelako (bere azpiprogrametan, noski, jokatu egingo du). Horrexegatik indefinitua uztea erabakitzen dugu. Hemendik aurrera, funtzioetan aipatzen ez diren kasuetarako definitu gabe daudela suposatuko dugu era berean.

Beste agindu batzuetan bigarren aldagai bat ere erabiltzen da. Azken hau atzitzeko honako funtzioa izango dugu:

2._aldagai_indize: **W** → **N**

eta bere esanahia hauxe da:

- 2._aldagai_indize(**XI:=cons$_{a_k}$(XJ);**) = j
- 2._aldagai_indize(**XI:=cdr(XJ);**) = j

cons funtzioan eta *car* predikatuan erabiltzen den ikurra (hitz moduan) aukeratzeko beste funtzio bat definituko dugu:

ikur_erabili: **W** → Σ^*

non:

- ikur_erabili(**XI :=cons$_s$(XJ);**) = '**s**'
- ikur_erabili(**if car$_s$?(XI) then P end if;**) = '**s**'

Mailarik altuena duen programaren azpiprogramari barne agindua deituko diogu. WHILE motako objektuekin lan egitean, programak azpiprogrametan deskonposatzeko honako funtzio hau maiz beharko dugu:

barne_agindu**: W → W**

while programa konposatu baten azpiprograma itzuliko diguna, eta beraz, oinarrizko programen gainean aplikatu ezin izango duguna:

- barne_agindu(**P$_1$ P$_2$**) = **P$_1$**
- barne_agindu(**if car$_s$?(XI) then P end if;**) = **P**
- barne_agindu(**while nonem?(XI) loop P end loop;**) = **P**

Baina programen konposaketak bi azpiprogramen bidez osatzen denez, beste eragiketa hau beharko dugu:

2._barne_agindu**: W → W**

soilik hurrengo kasurako definituta dagoena:

- 2._barne_agindu(**P$_1$ P$_2$**) = **P$_2$**

Programetan IKURKOP konstantea aipatuko dugu; IKURKOP alfabetoaren kardinalari dagokio. Funtzio eraikitzaileekin hasiko gara, adierazpide-funtzioari dagokionez inongo zailtasunik ez dutelako. Euren goiburuko amankomuna hauxe da:

```
package KONPUTAGARRIAK is
    type ARRUNTA is STRING;
    function kod_2 (STRING, STRING) return STRING;
    function kod_3 (STRING, STRING, STRING) return STRING;
    function "+" (ARRUNTA, ARRUNTA) return ARRUNTA;
end KONPUTAGARRIAK;
```

esleipen_hutsa_eratu:

```
X0 := kod_2(0, X1);
```

esleipen_cons_eratu

```
if nonem?(X2) then  X0 := kod_3(lehena(X2), X1, X3);
else X0 :=⊥⊥;
end if;
```

esleipen_cdr_eratu:

```
X0 := kod_3(IKURKOP+1, X1, X2);
```

konposaketa_eratu:

```
X0 := kod_3(IKURKOP+2, X1, X2);
```

baldintzazkoa_eratu:

```
if nonem?(X1) then  X0 := kod_3(IKURKOP+2+lehena(X1), X2, X3);
else X0 :=⊥⊥;
end if;
```

iterazioa_eratu:

```
X0 := kod_2(2*IKURKOP+3+X1, X2);
```

Ikuskatze funtzioak oso predikatu sinpleak dira. Izan ere, guztien while-errekurtsibitatea **2.** proposizioaren bidez froga izan genezakeen (beste predikatu errekurtsiboen eta funtzio konputagarrien konposaketak izanik), baina era homogeneoan egiteko programa-idazkeraz egingo dugu. Haien goiburuko amankomuna honako hau izango da:

```
package KONPUTAGARRIAK is
    type BOOLEARRA is STRING;
    type ARRUNTA is STRING;
    function deskod_2_1 (STRING) return STRING;
    function "<= " (ARRUNTA, ARRUNTA) return BOOLEARRA;
    function "∧" (BOOLEARRA, BOOLEARRA) return BOOLEARRA;
    function "+" (ARRUNTA, ARRUNTA) return ARRUNTA;
```

```
        function "*" (ARRUNTA, ARRUNTA) return ARRUNTA;
    end KONPUTAGARRIAK;
```

esleipen_hutsa_da?:

```
    X0 := deskod_2_1(X1) = 0;
```

esleipen_cons_da?:

```
    X0 := 1 <= deskod_2_1(X1) ∧ deskod_2_1(X1) <= IKURKOP;
```

esleipen_cdr_da?:

```
    X0 := deskod_2_1(X1) = IKURKOP + 1;
```

konposaketa_da?:

```
    X0 := deskod_2_1(X1) = IKURKOP + 2;
```

baldintzazkoa_da?:

```
    X0 := IKURKOP + 3 <= deskod_2_1(X1) ∧
                deskod_2_1(X1) <= 2 * IKURKOP + 2;
```

iterazioa_da?:

```
    X0 := 2 * IKURKOP + 3 <= deskod_2_1(X1);
```

Azkenik, atzipen-funtzioak inplementatu behar dira. Kasuka definitutako funtzioak izan arren programen bidez egingo dugu:

```
    package KONPUTAGARRIAK is
        type BOOLEARRA is STRING;
        type ARRUNTA is STRING;
        function deskod_2_1 (STRING) return STRING;
        function deskod_2_2 (STRING) return STRING;
        function "+" (ARRUNTA, ARRUNTA) return ARRUNTA;
        function deskod_3_2 (STRING) return STRING;
        function "*" (ARRUNTA, ARRUNTA) return ARRUNTA;
        function "–" (ARRUNTA, ARRUNTA) return ARRUNTA;
        function "<= " (ARRUNTA, ARRUNTA) return BOOLEARRA;
        function deskod_3_3 (STRING) return STRING;
    end KONPUTAGARRIAK;
```

aldagai_indize:

```
    case deskod_2_1(X1) is
    when 0 => X0 := deskod_2_2(X1);
    when 1 .. IKURKOP+1 => X0 := deskod_3_2(X1);
    when IKURKOP+2 => X0 := ;
    when IKURKOP+3 .. 2*IKURKOP+2 => X0 := deskod_3_2(X1);
```

```
when others  => X0 := deskod_2_1(X1) – 2 * IKURKOP – 3;
end case;
```

2._aldagai_indize:

```
if deskod_2_1(X1) >= 1 and deskod_2_1(X1) <= IKURKOP + 1 then
      X0 := deskod_3_3(X1);
else  X0 := ⊥⊥ ;
end if;
```

ikur_erabili:

```
case deskod_2_1(X1) is
when 1 .. IKURKOP => X0 := deskod_2_1(X1);
when IKURKOP+3 .. 2*IKURKOP+2 =>
                              X0 := deskod_2_1(X1) – IKURKOP – 2;
when others  => X0 := ⊥⊥ ;
end case;
```

barne_agindu:

```
case deskod_2_1(X1) is
when 0 .. IKURKOP+1 => X0 := ⊥⊥ ;
when IKURKOP+2 => X0 := deskod_3_2(X1);
when IKURKOP+3 .. 2*IKURKOP+2 => X0 := deskod_3_3(X1);
when others  => X0 := deskod_2_2(X1);
end case;
```

2._barne_agindu:

```
if deskod_2_1(X1) = IKURKOP+2 then
      X0 := deskod_3_3(X1);
else X0 := ⊥⊥ ;
end if;
```

Eragiketa hauen bidez while programekin ekintza benetan konplexuak programatzeko gai izango gara, besteak beste, euren egikaritzapena, trazak jartzea edota emaitzak maneiatzea. Hori txosten honetatik at geratzen da, baina bere jarraipena izango den beste lan batean hemen ezarritako oinarriak Konputagarritasunaren emaitza ezagunenak deskribatzeko aplikatuko dira.

5. Ondorioak

Lan hau, aldez aurreko prestakuntza teorikorik ez baina goi-mailako programazio lengoaiaren batean esperientzia duten lehen zikloko ikasleei zuzenduta dago. Aurreko kapituluak Konputagarritasun Teoriaren ikasketarako oinarri izan daitezen dugu helburu. Emaitza espezifikorik ematen ez den arren, aurkeztutako eredua ikasketa horretan erabili ahal izateko oinarrizko osagaiak ematen dira. Adibidez, Zenbagarritasun Teoremak, S-M-N edota Errekurtsio Teorema frogatzea (bereziki bigarren hau, Kleeneren Teorema izenaz ere ezaguna dena) oro har luze jotzen badu ere, testu honetan aurkeztutakoan oinarriturik froga bakoitzeko orri pare bat nahikoa izango dira (frogak ulertzeko egin beharreko ahalegina beste gai bat da, gure asmoa ez baita hori neurtzea). Nolabait Konputagarritasun Teoriaren "alde samurrena" biltzen saiatu gara, hau da, konputazioaren eredu orokor bat (kasu honetan while programak) era positiboan garatzen den aldea, bere ahalmena noraino iristen den egiaztatuz. Era berean, programazioa bere zentzu praktikoan garrantzitsua den zatia da, ikasleak oinarrizko kontzeptuak ezagun izango dituelarik.

Konputagarritasun Teoria oso disziplina zaharra da. Teknologikoki elektronika digitalean oinarritutako lehen konputagailuak agertu zirenerako (40. hamarkadaren bukaeran) oinarrizko aurrebaldintzak ezarrita zeuden. Hau da, konputagailuen muga nagusiak eraikiak izan aurretik ezagutzen zirela.

Antzinatasun honek, mirestekoa den arren, desabantailak ere baditu: hasiera batean konputaezintasuna frogatzeko erabili ziren ereduak eta notazioak orduz geroztik ia ez dira aldatu. Konputagailu abstraktuaren eredu erabiliena Turing-en Makina da dudarik gabe, eta konputazioaren prozesua zenbakien manipulazio gisa ikusten da testu klasiko gehienetan. Hau Gödel-en garaitik datorkigun ohitura bitxia da.

Injinerutza Informatikoko ikasle batentzat hori arazo bat izan ohi da. Batetik, Turing-en Makina programazioa erabat astuna egiten duen gailu arrotza da. Hori gutxi balitz, eredu teorikoaren eta (ezagutzen hasiak diren) konputagailu errealen arteko urruntasunak emaitzen inguruko eszeptizismoa eragiten du: "Turing-en Makina bat problema hau ebazteko gai ez dela frogatu dugun arren, nola jakin dezakegu beste gainerako konputagailuak ere ez direla gai izango? Zein erlazio dago bien artean?". Gainera, Informatika Teorikoarekin hasteko Automaten Teoria eta Lengoaia Formalei buruzko oinarrizko ikastaroa ematen badute, beste haustura bat ere gertatuko da: ikurren katea arbitrario baten gainean eragin eta

emaitza boolearra sortzen duten konputazio ereduak (automata finituak, piladunak edota lineal bornatuak) ikasten hasten dira eta, jarraitasun ebazpiderik gabe, eragiketa aritmetikoak egin eta emaitza arbitrarioa duten gailuetara pasatzen dira. Horrela, Lengoaien Teoriaren emaitzak Konputagarritasun Teoriaren aurrekari gisa integratzea ere zaila gertatzen da.

Lehenengo arazoa ebazteko eta konputagarritasuna prestakuntza erabat praktikoa duten pertsonengana hurbiltzeko saiakuntza batzuk egin dira. Zehazkiago, Cutland-en Erregistro Makinak edota Kfoury, Arbib eta Moll-en while programak Informatikaren kontzeptu abstraktuen irakaskuntzan programazio-esperientzia nola aprobetxatu daitekeeneko adibide onak dira. Baina tamalez bigarren arazoari (Automata Teoria eta Konputagarritasun Teoria uztartzeari) ez zaio horrenbesteko arreta eskaini. Horixe da testu honen helburua: While Programak birformulatzea, euren abantailetaz baliatuz baina konputazioa ikur arbitrarioen manipulazio gisa definituz; hau informatikaren errealitatetik askoz ere hurbilago dago.