TARTU ÜLIKOOL
Arvutiteaduse instituut

Härmel Nestra

SISSEJUHATUS FUNKTSIONAALSESSE PROGRAMMEERIMISSE

TARTU 2010

TARTU ÜLIKOOL
MATEMAATIKA-INFORMAATIKATEADUSKOND
ARVUTITEADUSE INSTITUUT

Härmel Nestra

SISSEJUHATUS FUNKTSIONAALSESSE PROGRAMMEERIMISSE

TARTU 2010

Kaane kujundanud Aita Linnas

Raamatu väljaandmist on toetanud Eesti Infotehnoloogia Sihtasutus programmi “Tiigriülikool+” raames.

Raamatu trükiversioon erineb elektroonilisest versioonist!

ISBN 978-9949-19-312-7

Tartu Ülikooli Kirjastus
www.tyk.ee
Tellimus nr 77

Sisukord

Eessõna

Käesolev raamat on välja kasvanud praktikumimaterjalidest Tartu Ülikooli informaatika ja infotehnoloogia erialade bakalaureuseõppe aines "Funktsionaalprogrammeerimise meetod", mille õpetamisega on autor olnud seotud aastast 2001. Tegu on funktsionaalse programmeerimise algkursusega.

Algul said igaks praktikumiks paberile pandud praktiliselt vaid märksõnad ja rida ülesandeid, hiljem lisandus järjest ka üldisi selgitusi. Selgitused omakorda tingisid uute, veel üldisemate selgituste lisamist. Nii on see tänaseks märkamatult arenenud valdkonda süstemaatiliselt avavaks ja rikkalikult näidetega katvaks *ca* 350-leheküljeliseks õpikuks. Ülesannete arv on kerkinud 500-ni (täpsus, muide, juhuslik).

Oma osa nii mahuka üllitise sünnis on ehk ka autori fanaatilisusel. Juba paar aastat tagasi mainis kraadiõppur ja osalt juba kolleeg Margus Niitsoo, et veel mõned aastad varem (kui tema seda ainet kuulas) olnud materjal juba täiesti korralik. Suurem osa mahtu on aga lisandunud pärast seda.

Et aine "Funktsionaalprogrammeerimise meetod" annab vaid 2 ainepunkti (3 Euroopa ainepunkti), siis muidugi ei ole käesolev õpik selles isegi mitte kõrgeima hinde saamiseks tervikuna kohustuslik. Teemad on õpikus kaetud liiaga, võimaldades erihuviga tudengitel jooksvalt ka sügavamale valdkonda tungida, rääkimata mitmest teemast, mis jäävad tervikuna väljapoole õppeaine sisu. Autori hinnangul vääriks kogu õpiku edukalt läbi töötanud tudeng vähemalt poolteisekordses mahus ainepunkte.

Õpetuse aluseks on nii siin õpikus kui ka aines programmeerimiskeel Haskell, mis on tänaseks kujunenud maailmas laisa funktsionaalse programmeerimiskeele n-ö esmavalikuks, musternäidiseks. Täpsemalt, õpikus on tuginetud standardile Haskell 98. Viimaste paranduste-täienduste tegemise ajal (novembris 2009) on keele kodulehele ilmunud teade, et järgmine standard, Haskell 2010, on valmis ja vähemalt üks seda täielikult toetav Has-

kelli realisatsioon (väidetavalt GHC 6.14.1) peaks kättesaadav olema oktoobriks 2010. (Osaliselt toetavad uut standardit juba praegu kõik tuntud Haskelli realisatsioonid.)

Käesoleva õpiku materjali suhtes muudab uus standard tegelikult üsna vähe. On ainult üks siin käsitletav keelekonstruktsioon — valvurikonstruktsioon —, mida uus standard tuntavalt üldistab, ja tegu on vaid programmeerimist teatud keerulisemates olukordades mõnevõrra lihtsustava võimalusega, midagi põhimõtteliselt uut see ei lisa. Peaaegu kõik ülejäänud muudatused uues standardis kas puudutavad mõnd süntaksi väga spetsiifilist detaili, millest selles õpikus nagunii mööda minnakse, või lisavad uusi võimalusi keele neisse algajale programmeerijale kaugetesse osadesse, millest siin õpikus juttu ei tehta. Üks üldistus — hierarhilised moodulid — on *de facto* standard olnud juba aastaid (kõik Haskelli realisatsioonid toetavad) ja see on raamatus sees.

Selle järgi, et raamat on mõeldud eelkõige õpikuks kursusel, mida tudengid üldjuhul võtavad kolmandal õppeaastal, on valitud ka eelteadmiste hulk, millele selgitused tuginevad. See tähendab, et kõigist selgitustest arusaamiseks on vajalik mõningane varasem programmeerimiskogemus mingis imperatiivses keeles ja tutvus programmeerimise põhimõistetega. Samuti on soovitav tugev matemaatiliste teadmiste tase orienteeruvalt ülikooli algebra algkursuse ja muidugi ka keskkooli matemaatikaprogrammi ulatuses.

Sellise raamatu kirjutamine on suurem töö kui autor iganes arvas. Kui oleks aimanud, et raamat nii mahukas tuleb, ei oleks küll nõnda kartmatult nii põhjalikuks läinud. Autor tänab professor Varmo Venet moraalse toetuse eest.

Soovigem lugejale jaksu, indu ja avastamisrõõmu!

1 Sissejuhatus

1.1 Funktsionaalne programmeerimine

Ajalooliselt välja kujunenud programmeerimisstiilid erinevad üksteisest fundamentaalselt. Loomuldasa kokkukuuluvatest põhimõtetest koosnevat terviklikku programmeerimisstiili nimetatakse programmeerimisparadigmaks. Funktsionaalne programmeerimine on üks neist.

Programmeerimiskeeli võib liigitada selle järgi, millises paradigmas töötamiseks nad on mõeldud; näiteks on olemas funktsionaalsed, loogilised, objektorienteeritud, protseduurorienteeritud jne programmeerimiskeeled. Tänapäeval toetavad paljud levinud keeled mitut paradigmat.

Sellised on näiteks Ocaml ja Python.

Paradigmad moodustavad hierarhia, kus funktsionaalne programmeerimine liigitub koos loogilise programmeerimisega deklaratiivse programmeerimise alla, vastandudes imperatiivsele programmeerimisele.

Kui imperatiivses programmeerimises kirjeldab kood eeskätt arvutusprotsessi käiku, siis deklaratiivses programmeerimises kirjeldab programmikood esmajoones hoopis matemaatilisi objekte ja abstraktseid seoseid nende vahel. Loomulikult vastab deklaratiivses programmeerimiskeeles kirjutatud korrektsele koodile ka mingi arvutusprotsess, mis koodi jooksutamisel toimub, kuid see jääb programmeerija jaoks tagaplaanile.

Kui imperatiivses keeles kirjutatud koodi mõte selgub kirjapandud arvutussammudest, siis deklaratiivne paradigma võimaldab esitada abstraktsed seosed arvutusprotsessi osaliste — näiteks sisendi ja väljundi — vahel otse, ilma arvutusprotsessi detailide abita. Nii saab deklaratiivses keeles kirjutatud koodi mõtet suures osas mõista ka täiesti ilma tähelepanu pööramata

sellele, kuidas masin selle järgi asju reaalselt arvutaks.

Distantseerumine arvutusprotsessi detailidest on üks modulaarsuse vorm. Modulaarsus programmeerimises tähendab üldiselt igasugust erilaadse info selget üksteisest lahushoidmist koodis. Eri vaadete, erilaadse info hoidmine eraldi võimaldab inimesel kergemini neid vaateid ja kogu koodi mõista ja vähendab koodi hooldusega seotud töömahtu. Modulaarsuse mitmeid vorme realiseerivad ka tänapäeva tuntud imperatiivsed programmeerimiskeeled.

Muidugi peab ka deklaratiivses keeles programmeerija reaalse arvutuse käiguga arvestama. Sama abstraktset funktsiooni on võimalik arvutada kiiremini või aeglasemalt ja pole kasu koodist, mis kirjeldab küll õiget seost, kuid mille järgi arvutamine võtab rohkem aega kui Universumi eluiga.

Kuigi ajalooliselt ilmusid imperatiivne ja deklaratiivne paradigma enamvähem samaaegselt (esimese imperatiivse programmeerimiskeelena Turingi masin 1936, esimese deklaratiivse programmeerimiskeelena lambdaarvutus samuti 1936), on tänapäeval imperatiivsed keeled levinumad ja kasutatavamad kui deklaratiivsed.

1.1.1 Olemuslikud omadused

Käsitleme siin lähemalt tunnuseid, mis on omased eelkõige funktsionaalsele paradigmale. Need on konkreetsed teed saavutamaks deklaratiivse programmeerimise põhieesmärki esitada abstraktseid seoseid matemaatiliste objektide vahel.

1. Avaldiste väärtustamine. Esimene tunnus on avaldiste väärtustamise keskne roll arvutusprotsessis. Avaldise väärtustamiseks nimetatakse avaldise teisendamist eesmärgiga saada infot tema väärtuse kohta.

Arvutus, mida tahetakse programmeerida, esitatakse avaldisena ja masina töö seisneb selle avaldise väärtustamises ja täitmises. Tüüpilisest funktsionaalsest programmist moodustavad suurema osa definitsioonid, mis kirjeldavad abstraktseid seoseid andmete vahel, ühtlasi aga kujutavad endast avaldiste väärtustamise reegleid, mille järgi masin neid teisendab.

2. Funktsioonid kui andmed. Funktsionaalses programmeerimses kasutatakse andmestruktuure, funktsioone ja protseduure võrdsel alusel üksikandmetega; funktsionaalses keeles on nad kõik süntaktiliselt eristamatud.

Funktsioonid on andmed nagu arvudki, nad saavad esineda samades rollides. Näiteks võib iga funktsioon olla avaldise väärtuseks ja teise funktsiooni argumendiks, samuti esineda andmestruktuuri komponendina. Selle kohta öeldakse, et funktsioonid on esimese kategooria objektid. Sama kehtib ka andmestruktuuride ja protseduuride kohta.

Et protsesse saaks käsitada abstraktsete andmetena, peavad olemas olema funktsioonid, mis väiksematest protsessidest panevad kokku suuremaid. See võimaldab soovitud arvutusprotsesse esitada avaldise kujul. Avaldise "väärtustamine ja täitmine", millele ülal oli viidatud, tähendab avaldise väärtustamist ja väärtuseks oleva protsessi läbiviimist.

3. Ilmutatud viidatavus. Edasi pöörame tähelepanu sellisele väga olulisele tunnusele, mida eesti keeles on nimetatud ilmutatud viidatavuseks. See sätestab, et alamavaldis mõjutab kogu avaldise väärtuse kujunemist alati ainult oma väärtusega. See tähendab, et alamavaldist võib alati asendada sama väärtusega avaldisega, tulemus sellest ei muutu.

Näiteks kui `log` tähendab naturaallogaritmi ja aritmeetilised avaldised `1 + 1` ja `2` on oma tavapärase tähendusega (seega mõlemad väärtusega 2), siis ilmutatud viidatavuse põhjal on `log (1 + 1)` ja `log 2` ühesuguse väärtusega.

Ilmutatud viidatavus võib esmapilgul tunduda enesestmõistetav. Matemaatikas on ta üheks alusprintsiibiks ("võrdsete omavahelisel asendamisel saame võrdsed"), filosoofid nimetavad sarnast printsiipi Leibnizi seaduseks ("võrdsetel on kõik omadused ühised"). Samas imperatiivses programmeerimises ei saa ilmutatud viidatavuse kehtimisest üldiselt juttugi olla juba seetõttu, et muutujate väärtused on vabalt muudetavad.

Imperatiivses keeles võib mingi funktsioon f muuta mõne globaalse muutuja x väärtust ja teine funktsioon g seda muutujat arvutuses kasutada. See, et `f ()` väärtus on näiteks 1, ei tähenda, et `f ()` ja `1` on vastastikku asendatavad. Avaldised `g (f ())` ja `g (1)` võivad olla erineva väärtusega, sest teisel juhul arvutab g oma väärtuse x algse väärtuse põhjal, kuid esimesel juhul f poolt muudetud väärtuse põhjal.

Nähtust, kus avaldise väärtustamine toob kaasa mingi olekumuutuse, mis võib edaspidi leitavaid väärtusi mõjutada, nimetatakse kõrvalefektiks.

Äsjatoodud näites oli muutuja x väärtuse muutumine funktsiooni f kõrvalefekt.

Niisiis on ilmutatud viidatavus teisiti öeldes kõrvalefektide puudumine

avaldiste väärtustamisel. Väärtustamine on absoluutselt läbipaistev; kui meid huvitab ainult tulemus, võib arvutuse detailid arvestusest välja jätta.

Tegelikult puuduvad kõrvalefektid funktsionaalse programmi puhul täiesti, sest kogu tegevus on kirjeldatud avaldiste kaudu.

Haskell — keel, millega selles õpikus töötatakse, — on selles mõttes puhtalt funktsionaalne, et ta mitte ainult et toetab programmeerimist funktsionaalses paradigmas, vaid temas teisiti ei saagi programmeerida. Muuhulgas on ilmutatud viidatavus keele ülesehituse poolt garanteeritud. Sellel on kolossaalsed järeldused keele võimaluste kohta.

Ühegi muutuja väärtus ei saa kunagi sama ploki samal lugemisel muutuda, muidu saaks seda muutujat kasutada kõrvalefekti tekitamiseks. Muutujad pole muutujad samas mõttes nagu tavaks imperatiivses programmeerimises. Funktsiooni erinevatel väljakutsetel võivad formaalsed parameetrid omandada küll erinevaid väärtusi, kuid funktsiooni väärtuse leidmise käigus nad enam muutuda ei saa. See on nagu matemaatikas: valemis esinevad muutujad võivad küll omandada mitmeid väärtusi, kuid valemi samal lugemisel on sama muutuja kogu valemi ulatuses sama väärtusega.

Näiteks valem $x^2 = x \cdot x$ on õige ainult tänu sellele, et samal lugemisel on x igal pool ühe ja sama väärtusega.

Ka keerulisemad avaldised on ploki samal lugemisel alati sama väärtusega. Isegi välise maailma sündmustel (klahvivajutused jms) ei tohi olla avaldiste väärtustele mingit mõju. Alalises muutumises olev väline maailm on puhastest, sündmustest mõjutamata andmetest jäigalt eraldatud nagu džinn pudelis. See on samuti üks modulaarsuse vorm.

Ilmutatud viidatavuse puhul saab funktsiooni sisendi igale võimalikule väärtusele vastata ainult üks väljundi väärtus, ehk programmeerimiskeele funktsioon määrab matemaatilise funktsiooni matemaatilistel objektidel.

On olemas poolfunktsionaalseid keeli nagu Standard ML, kus kehtivad siin loetletud kaks esimest funktsionaalse keele tunnust ning ka muutujate väärtused ei muutu, kuid funktsioonid väljendavad pigem algoritme, mis võivad tulemuse arvutamisel kasutada välist, muutuvat infot, nad pole matemaatiliste funktsioonidena interpreteeritavad. Ilmutatud viidatavus neis keeltes ei kehti.

Et muutuja väärtust muuta ei saa, pole funktsionaalses keeles omistamist ja puuduvad ka imperatiivsetest keeltest tuttavad tsüklikonstruktsioonid. Selle kitsenduse tähendust võib mõista ka vaid imperatiivseid keeli tundes: prog-

rammeerida kogu arvutus funktsiooniväljakutsete peal ja sealhulgas kõik tsüklilised protsessid rekursiooniga. (Imperatiivses keeles kitsendataks nii programmeerimisvabadust tegelikult rohkem kui funktsionaalne keel seda teeb, sest viimases on neile piiranguile vastukaaluks funktsioonid esimese kategooria objektid.)

Ilmutatud viidatavus ühelt poolt küll ahendab võimalusi algoritmi valikuks, kuid tulemusena paraneb programmide loetavus.

4. Muud tunnused. Tihti peetakse funktsionaalse paradigma tunnusteks veel üht-teist, mis siiski paljudes funktsionaalsetes keeltes puuduvad, näiteks ülitugevat tüübisüsteemi või laiska väärtustamist. Haskellis on viimatimainitud omadused olemas, nii et nendega teeme nii või teisiti tutvust.

Programmeerimiskeelega määratud kõikvõimalike andmete, avaldiste jne klassifikatsiooni tüüpidesse koos tüüpide omaduste ja omavaheliste suhete võrgustikuga nimetatakse tüübisüsteemiks. Haskelli tüübisüsteem on nii hea, et enamasti on koodis esinevate avaldiste tüübid automaatselt kindlaks tehtavad, programmeerija ei pea tüüpe ise deklareerima. Igal juhul tehakse avaldiste tüübid kindlaks enne programmi töö algust kompileerimise ajal ja mida rangem tüübisüsteem, seda suurema tõenäosusega leitakse programmeerimisviga juba siis ja vigane kood ei lähe kunagi käiku.

Laisk väärtustamine tähendab väärtustamist vastavalt vajadusele: kui mõni funktsioon oma väärtuse leidmisel argumenti ei kasuta, siis argumenti ei väärtustatagi, ja komplekseid argumente (andmestruktuure) väärtustatakse ainult sellisel määral, millisel funktsioon neid vajab. See teenib tööaja kokkuhoiu eesmärki. Samas on see kahe teraga mõõk, sest laisa väärtustamise korral kipub mälu arvutuse käigus täituma pikkade väärtustamata avaldistega, mis väärtustatuna nõuaksid palju vähem ruumi. Laisale väärtustamisele vastandub agar väärtustamine, mis tähendab, et funktsiooni väljakutsel kõigepealt väärtustatakse argument.

Kuigi mitmetes levinud funktsionaalsetes keeltes tüübisüsteem puudub ja väärtustamine on agar, on ülitugeva tüübisüsteemi ja laisa väärtustamise pidamine funktsionaalse paradigma omadusteks õigustatud selles mõttes, et neist on rohkem kasu just funktsionaalse paradigma kontekstis. Imperatiivsetes keeltes tüübisüsteeme nii tugevaks ei arendata ja laisad on tavaliselt vaid loogilised operatsioonid, kõik omadefineeritud funktsioonid käituvad agaralt. Kui funktsionaalses keeles ei mõjuta väärtustamisstrateegia valik ilmutatud viidatavuse tõttu programmi mõtet, siis ilmutatud viidatavuse puudumisel muudaks ohjeldamatu laisk väärtustamine programmide

mõtte raskemini tabatavaks.

1.1.2 Eelised ja puudused

1. Eelised. Funktsionaalse programmeerimise tugeva küljena mainime kõigepealt koodi suurt väljendusvõimsust ja paindlikkust erinevate ülesannete lahendamisel. Tänu sellele, et funktsioonid on esimese kategooria objektid ja iga funktsiooni saab kasutada teise funktsiooni argumendina, on arvutused piiranguteta esitatavad parameetrilisena teise arvutuse suhtes. See võimaldab probleemilahendused viia abstraktsemale tasemele kui tavalistes programmeerimiskeeltes. Tihti saab ühe ja sama koodijupiga realiseerida arvutusi väga erinevat laadi andmetega, mis vähendab koodikordusi.

Arvutuste spetsifitseerimine abstraktsel tasemel vähendab vajadust arvutusprotsessi detailide kallal nokitsemise järele. Kood tuleb lühem ja ülevaatlikum kui imperatiivsetes keeltes. See tähendab programmeerija tööaja vähenemist sama ülesande lahenduse programmeerimisel võrreldes imperatiivse programmeerimisega. Vilunud programmeerija on võimeline probleemilahenduse esimese töötava versiooni (mis ei pruugi küll olla kõige efektiivsem) produtseerima väga kiiresti.

Ericssoni kogemus näitab, et üleminek imperatiivselt keelelt funktsionaalsele mobiiltarkvara arendamisel kiirendas programmeerijate töökiirust kuni kümme korda.

Ilmutatud viidatavus võimaldab koodi mõista matemaatikast tuttaval moel, see lihtsustab koodist arusaamist ja võimaldab programmi mõtte tabamist ilma arvutuskäigu detailidesse laskumata. Juhtudel, kus programmi korrektsus on nii tähtis, et tuleb kõne alla korrektsuse formaalne tõestamine, teeb ilmutatud viidatavus ja süntaksi-semantika matemaatikalähedus selle töö palju lihtsamaks ja inimlikumaks kui see oleks imperatiivse keele puhul. Veenduda tuleb vaid soovitavate abstraktsete seoste kehtimises, arvutuse korrektsus tuleb sellega tasuta kaasa (reaalse arvutuse korrektsus sõltub muidugi veel kasutatava kompilaatori või interpretaatori korrektsusest, aga neid on programmeerija nii või teisiti sunnitud usaldama).

2. Puudused. Funktsionaalse programmeerimise esimese puudusena võib tuua koodi ressursinõudlikkuse võrreldes imperatiivsetes keeltes kirjutatud koodiga ja seda nii aja- kui mälutarbe osas. Mida enam on keel funktsionaalne, seda enam aega võtavad arvutused neis kirjutatud programmidega

ja seda enam söövad neis kirjutatud programmid mälu (viimane reegel kehtib nii täitmisaegse mälutarbe kui ka kompileerimisel saadud masinakeelse programmi mahu kohta). Ressursinõudlikkus tuleneb esmajoones probleemidest funktsioonide käsitlemisel esimese kategooria objektidena. Ka on funktsiooniväljakutsete mehhanism ise arvutuslikult võrdlemisi kohmakas.

Programmeerimises näib kehtivat kuldreegel: mida vähem kulutame ressursse programmide loomisel, seda rohkem kulutame ressurssi nende kasutamisel. Funktsionaalne keel võimaldab olulist säästu programmeerija tööajas, kuid funktsionaalses keeles kirjutatud programmid jooksevad suurusjärke aeglasemalt. Oleneb konkreetsest olukorrast, kumb kulu on olulisem. Kui programmi töökiirus on esmatähtis, tuleb funktsionaalsest paradigmast paraku loobuda. Kui programmi töökiirus on teisejärguline, on funktsionaalne paradigma hea valik. Võib loota, et seoses protsessorite kiiruse pideva suurenemisega kasvab nende ülesannete hulk, mille puhul tippkiirus pole enam vajalik, ja ühtlasi funktsionaalse paradigma läbilöögivõime.

Teine funktsionaalse programmeerimise puudus seondub tema abstraktsusega. Et garanteerida ilmutatud viidatavus, peab mäluhaldus, st arvuti mälu reserveerimine programmi töö tarbeks ja ärakasutatud mälu vabastamine (viimast nimetatakse prügikoristuseks), toimuma automaatselt, sest programmeerija võib mälule otsese ligipääsu korral andmed kergesti rikkuda. Kuid mõnikord oleks otsene mäluhaldus vajalik, sellisel juhul ei saa funktsionaalset keelt tööks kasutada.

Funktsionaalse programmeerimise kitsaskohaks peetakse tihti üldisemalt igasuguse interaktiivse protsessi programmeerimist, st selliste arvutuste programmeerimist, kus pidevalt tuleb võtta keskkonnast uut infot ja seda sinna produtseerida. Kui imperatiivses keeles käib infovahetuse programmeerimine keeles loomuldasa olemasolevate vahenditega, siis funktsionaalses keeles seab ilmutatud viidatavus sellele piirangud. Programmeerija ei saa keskkonnast loetud infot lihtsalt niisama kuhugi muutujasse omistada. Lisaks nõuab funktsionaalne keel protsessi väljendamist mingi avaldise väärtusena, imperatiivses keeles pole vaja sellist tsirkust etendada.

Mis puutub Haskelli, siis selles on niisuguste avaldiste kirjutamiseks välja töötatud spetsiaalne süntaks, mis võimaldab interaktiivseid tegevusi programmeerida peaaegu niisama ladusalt kui mistahes imperatiivses keeles. Seega võib öelda, et siin on tegemist juba praktiliselt ületatud probleemiga.

1.2 Haskelli maailm

Enne kui tutvume Haskelli süntaksiga ja jõuame esimeste funktsionaalsete programmide kirjutamiseni, anname lühiülevaate Haskellist just semantika poole pealt: visandame pildi matemaatiliste objektide maailmast, milles Haskell-koodi tuleb mõista. Ehkki need objektid sarnanevad vägagi matemaatikast tuttavate struktuuridega, tuleb mitmeid mõisteid tunda nende tähenduses, mis on vastava matemaatika mõiste omast pisut erinev. Olles selle maailma asukatega eelnevalt tuttav, on süntaksi õppimine ja süntaktiliste konstruktsioonide tähenduse tabamine hõlpsam.

Arvestagem järgnevas, et nende asukate nimetamine **objektideks** ei tähenda paralleeli tõmbamist objektidega objektorienteeritud paradigma spetsiifilises mõttes. Vastupidi, ilmutatud viidatavuse tõttu puudub funktsionaalses keeles objekti identiteet, nagu seda tuntakse objektorienteeritud programmeerimises. Täpsemalt, objekti identiteet ja olek pole funktsionaalses paradigmas eristatavad — olek identifitseeribki objekti. (Objekti identiteeti koos muutuva olekuga on võimalik lavastada, näiteks lugedes objekti oleku välise maailma sündmuseks, aga seda me lähemalt vaatlema ei hakka.)

Kõige jämedamas plaanis jagunevad Haskelli maailma objektid andmeteks, tüüpideks, liikideks ja klassideks. Tüüpidel, liikidel ja klassidel on klassifitseeriv roll. Tüübid klassifitseerivad andmeid, liigid ja klassid omakorda tüüpe. Järgnev tutvustav ringkäik on üles ehitatud nende klassifitseerimisvahekordade kaupa.

1.2.1 Andmed ja tüübid

Nagu juba funktsionaalse paradigma kirjelduses juttu oli, peetakse selles paradigmas väga erinevaid asju **andmeteks**[1]. Haskellis teevad erinevat laadi andmete vahel vahet **tüübid**. Tüübisüsteem on väga rikas.

1. Loetelutüübid. Tüüpi `Int` kuuluvad 4-baidised märgiga **täisarvud**. Vähim täisarv tüübis `Int` on -2147483648, suurim on 2147483647. On

[1] Eesti keele sõnastikud väidavad, et sõna **andmed** pole ainsuses kasutatav, väljaarvatud erimõistetes nagu **andmebaas**. Astume siin sellest keelust üle, kuna sõnal **andmed** on siin tavalisest erinev, spetsiifilisem tähendus ning ainsuslik kasutus, erinevalt näiteks sõnadest **püks** ja **käär**, täiesti mõttekas. Alternatiiv oleks kasutada sõna **väärtus**, kuid see on niigi juba üle koormatud. Kuna tüübi kohta on üldkasutatav ka sõna **andmetüüp**, siis on igati loogiline, et tüüpidesse kuuluvad ikkagi andmed.

ka teine täisarvutüüp — Integer, mis arvudele suuruspiiranguid ei sea. Tüüp Integer on niisiis põhimõtteliselt lõpmatu. Tüübid Float ja Double on kaks erinevat **ujukomaarvu**tüüpi. Arvud neis tüüpides võivad erineda oma täpsuse poolest, ehk teisi sõnu, nende esitamisele kuluv baitide arv võib erineda. Mõlemas tüübis on see siiski fikseeritud, ujukomaarvud ei saa minna kuitahes suureks. Täpne baitide arv sõltub realisatsioonist, aga kehtib tingimus, et tüübi Double arvud peavad olema vähemalt niisama täpsed kui tüübi Float omad. Tüüp Char hõlmab **sümbolid** (mitte ainult 1-baidised).

Tüüp Bool sisaldab **tõeväärtused**, mida tähistame False ja True; esimene tähendab väära ja teine tõest. Mõneti sarnane on tüüp Ordering — **suurusvahekorra**tüüp. Andmed sellest tüübist tähistavad võimalikke tulemusi kahe võrreldava andme võrdlemisel. Võimalikud tulemused on "väiksem", "niisama suur", "suurem", mida tähistame vastavalt LT, EQ, GT.

Kõik senivaadeldud tüübid on nn loetelutüübid. Tüüpi nimetatakse **loetelutüübiks**, kui temasse kuuluvad vaid üksikandmed, st ükski anne selles tüübis ei koosne omakorda teistest. Nimetus tuleneb sellest, et loetelutüüpe pole võimalik lihtsamalt formaalselt defineerida kui loetledes kõik temasse kuuluvad andmed.

Reaalses elus ei pruugi arvutus lõppeda normaalse tulemusega. Hõlmamaks ka n-ö halbu juhte, loetakse iga tüüp sisaldavaks peale normaalsete andmete ka nn **bottomit**, mida tähistatakse sümboliga $\perp$. Koos bottomiga sisaldab näiteks Bool kolme ja Ordering nelja annet. Bottom tähendab normaalsel kujul info täielikku puudumist.

Näiteks täisarvude 1 ja 0 jagamisel on tulemuseks $\perp$.

2. Struktuuritüübid. Loetelutüüpidele vastanduvad struktuuritüübid, milles sisalduvad **andmestruktuurid**. Kuid ka andmeid, mis moodustavad loetelutüübi, on mugav lugeda triviaalseteks andmestruktuurideks, mis kunagi midagi ei sisalda.

2.1. Listitüübid. List on üht tüüpi andmete kogum, kus andmed on paigutatud ühte jorru. Listis paiknevaid andmeid nimetatakse listi **elementideks**. List on Haskellis ja üldse funktsionaalses programmeerimises põhilisimaks kasutatavaks andmestruktuuriks.

Lihtsaim list on **tühi** list, milles pole ühtki elementi; tähistagu seda kirjutis []. Mittetühjad listid avalduvad operatsiooni : abil, mis tähendab ühe elemendi lisamist listi algusse; seejuures kui kirjutame $a : b : l$, siis mõtleme

sama mis kirjutades $a : (b : l)$.

Näiteks $1 : []$ on üheelemendiline täisarvude list. Samamoodi on täisarvude listid ka näiteks $0 : 1 : []$ (kaheelemendiline), $13 : 11 : 2009 : []$ (kolmeelemendiline), $1 : 3 : 5 : 7 : 9 : []$ (viieelemendiline). Aga näiteks $\mathrm{True} : [], \mathrm{False} : [], \mathrm{True} : \mathrm{False} : []$, $\mathrm{False} : \mathrm{False} : \mathrm{False} : \mathrm{False} : []$ on tõeväärtuste listid.

Üldiselt, kui a on anne ja l on a-ga sama tüüpi andmete list, siis $a : l$ on sama tüüpi andmete list, kus esimene element on a ja ülejäänud elemendid on parajasti l elemendid samas järjekorras. Listis $a : l$ nimetatakse elementi a listi **peaks** ja alamstruktuuri l listi **sabaks**. Kui l on mingi list, siis l **alamlistideks** nimetatakse listi l ja, kui ta on mittetühi, ka l saba kõiki alamliste.

On oluline rõhutada, et Haskelli list ei pea olema lõplik. On olemas **lõpmatud listid**, mis ei sisalda alamstruktuurina tühja listi ning listi saba on niisama lõpmatu kui list ise. Kuid on veelgi liste, mis alamstruktuurina tühja listi ei sisalda: need, mille mõni alamlist on bottom. Selliseid nimetatakse **osalisteks listideks**. Laisk väärtustamine teeb lõpmatud ja osalised listid omaette andmetena mõttekaks.

Kõik eespool toodud näitelistid on lõplikud.

Lõpmatud listid on näiteks $0 : 1 : 2 : 3 : \ldots$, $\mathrm{False} : \mathrm{False} : \mathrm{False} : \mathrm{False} : \ldots$ jms.

Osalised listid on näiteks $1 : \bot$, $\mathrm{True} : \mathrm{False} : \mathrm{False} : \bot$ ja $\bot$ ise. Seejuures näiteks $\bot$ ja $1 : \bot$ erinevad teineteisest: esimeses puudub struktuur täiesti, teises aga on eristatavad pea ja saba.

Pangem lisaks tähele, et listi lõplikkus, lõpmatus või osalisus ei sõltu mingilgi määral listi elementidest, vaid ainult struktuurist.

Näiteks $\bot : []$ on täiesti lõplik list.

Kõik listid jagunevad tüüpidesse vastavalt elementide tüübile; ei ole olemas üht listitüüpi, kuhu kuuluksid kõik listid. Mistahes tüübi A korral seda listitüüpi, millesse kuuluvad listid elementidega tüübist A, tähistame $\mathrm{List}\, A$. Et elemendituup võib ise olla üks listitüüpidest, saame nii konstrueerida lõpmata palju Haskelli tüüpe.

Näiteks listid elementidega tüübist Int moodustavad tüübi $\mathrm{List}\,\mathrm{Int}$. Listid, mille elementideks on listid tüübist $\mathrm{List}\,\mathrm{Int}$, moodustavad tüübi $\mathrm{List}\,(\mathrm{List}\,\mathrm{Int})$. Nii saab jätkata kaugele tahes.

Sõne on Haskellis sama mis sümbolite list. Niisiis tüüp List Char on ühtlasi sõnetüüp. Vastavus on järgmine: tühi sõne kujutab endast tühja listi; mittetühja sõne pea on tema esimene sümbol ning saba tema alamsõne alates teisest sümbolist nii kaugele kui annab. Seega on Haskellis olemas lõpmatud ja osalised sõned.

2.2. Nurjumisega tüübid. Kuigi Haskellis on igas tüübis olemas bottom, mis tähendab oodatava normaalse tulemuse puudumist, ei ole see alati piisav vahend mugavaks ümberkäimiseks potentsiaalselt ebaõnnestuva arvutusega. Bottomi alla kuuluvad ka kõige hullemad ebaõnnestumised, mille esilekerkimist ei ole kunagi võimalik üldse kindlaks teha, nagu mõtlema jäämine lõpmata kauaks. Et võimaldada oodatava tulemuse puudumist kujutada normaalse andmena, on olemas n-ö nurjumisega tüübid Maybe A.

Kui A on mingi tüüp, siis Maybe A on tüüp, mis sisaldab kõiki tüüpi A kuuluvaid andmeid a kujul Just a ja lisaks spetsiaalset annet Nothing, mis tähendab arvutuse **nurjumist** selles mõttes, et tavapärane tulemus tüüpi A puudub.

Näiteks tüüp Maybe Int sisaldab muuhulgas andmeid Just 0, Just 1, Just (-18) ja Nothing, tüüp Maybe Bool koosneb aga andmetest kujul Just True, Just False ja Nothing (rohkem andmeid peale bottomi pole).

Nurjumisega tüübist võib tinglikult mõelda ka kui listitüübist, kus listid ei saa sisaldada üle ühe elemendi.

2.3. Korrutistüübid. Palju kasutust leiavad **järjendid**, mis on sarnased järjendite ehk vektoritega matemaatikas. Järjenditüüpide kohta öeldakse ka **korrutistüüp**, sest kui järjenditüübi bottom kõrvale jätta, siis on matemaatiliselt tegemist hulkade otsekorrutisega.

Korrutistüüpide alla kuuluvad paaritüübid, kolmikutüübid jne. Kui A ja B on tüübid, siis $A \times B$ on tüüp, mis sisaldab andmete **paarid** $(a\,, b)$, kus a tüüp on A ja b tüüp on B. Samamoodi $A \times B \times C$ sisaldab **kolmikuid** $(a\,, b\,, c)$, kus a tüüp on A, b tüüp B ja c tüüp C.

On olemas **ühiktüüp**, milles on ainult üks normaalne anne, mida tähistame (). Ühiktüüpi võib pidada korrutistüübiks, kus tegurtüüpide arv on 0.

2.4. Summatüübid. Korrutistüüpidele vastanduvad **summatüübid**. Kui A ja B on tüübid, siis Either A B on tüüp, mis sisaldab andmed kujul Left a ja Right b, kus a tüüp on A ja b tüüp on B. Summatüübi nimetus tuleb sellest, et kui bottomeid mitte arvestada, on andmeid tüübis Either A B samapalju

kui tüüpides A ja B kokku.

2.5. Võrdlus matemaatikaga. Andmestruktuuride käsitlus on seega vägagi matemaatikapärane, tuletades meelde üldist algebrat. Täieliku vastavuse matemaatikaga rikuvad ära bottomid, mida kõik struktuuritüübid sisaldama peavad, kuid millel pole mingit struktuuri.

Näiteks paaritüübi bottomit võidakse Haskellis küll paariks nimetada, kuid matemaatilises mõttes ta paar ei ole. Arvutades paaritüübi bottomi esimest või teist komponenti, saame mõlemal juhul tulemuseks bottomi (siis vastavalt esimese ja teise komponenttüübi oma), sest kust midagi võtta ei ole, sealt ei võta Haskell ka. Samas kuulub paaritüüpi ka $(\bot, \bot)$, mis on paar nagu muiste, kuid annab samuti komponentide arvutamisel tulemuseks bottomid. Siit on näha, et matemaatika seadus, mille kohaselt paarid on võrdsed, kui nende vastavad komponendid on võrdsed, siin alati ei kehti.

Analoogselt võime matemaatikas kehtiva võrduskriteeriumi paikapidamatust näidata kõigi struktuuritüüpide korral.

3. Funktsioonitüübid.

3.1. Funktsioonid. Andmestruktuuridest hoopis erinevat laadi objektid on funktsioonid. **Funktsioon** saab mingeid objekte **argumendiks** ja annab nende põhjal välja mingeid (üldjuhul uusi) objekte — **funktsiooni väärtusi**. Argumendi andmist funktsioonile nimetatakse **funktsiooni rakendamiseks** argumendile. Kui f on funktsioon ja x talle sobiv argument, siis f rakendamise tulemust argumendile x (ehk funktsiooni väärtust sel argumendil) tähistame $f\,x$.

Iga funktsioon võtab argumente ühest kindlast tüübist ja annab tulemusi ka ühest kindlast tüübist. Funktsioonid jagunevad tüüpidesse vastavalt oma argumendi- ja väärtusetüübile. Kui A ja B on suvalised tüübid, siis $A \to B$ on tüüp, mis koosneb funktsioonidest, mille argumenditüüp on A ja väärtusetüüp B.

Näiteks funktsioon, mis teisendab sümboli tema täisarvkoodiks, on tüüpi $\mathrm{Char} \to \mathrm{Int}$, naturaallogaritm on tüüpi $\mathrm{Float} \to \mathrm{Float}$ või $\mathrm{Double} \to \mathrm{Double}$ (st on kaks naturaallogaritmfunktsiooni, üks kummagi ujukomaarvutüübi jaoks) jne. Nurjumisega tüüpide juures tutvusime funktsioonidega Just, mis on tüüpi $A \to \mathrm{Maybe}\,A$ iga tüübi A jaoks.

Andmete omadusi ehk neile seatavaid tingimusi saab väljendada **predikaatidena**; nii nimetatakse funktsiooni, mille väärtusetüüp on tõeväärtu-

setüüp. Predikaadi väärtus True vastab juhtudele, kus tingimus kehtib.

Kuna funktsionaalses keeles võib iga funktsioon olla funktsiooni argument, samuti väärtus mingil argumendil, võivad funktsiooni argumenditüüp ja väärtusetüüp olla omakorda funktsioonitüübid. Pikemate funktsioonitüüpide kirjutamisel võib kokkuleppeliselt paremalt sulud ära jätta: $A \to B \to C$ tähendab sama mis $A \to (B \to C)$.

Näiteks tüüpi (Integer $\to$ Integer) $\to$ (Bool $\to$ Char) kuuluvad parajasti funktsioonid, mis võtavad argumendiks täisarve teisendavaid funktsioone ja annavad väärtuseks funktsioone, mis omakorda võtavad argumendiks tõeväärtusi ja annavad tulemuseks sümboleid. Selle funktsioonitüübi võib kirjutada ka kujul (Integer $\to$ Integer) $\to$ Bool $\to$ Char.

3.2. Karritamine. Haskellis pole mitme muutuja funktsioone. Sõltuvused mitmest parameetrist tuleb esitada raamistikus, millega äsja tutvusime. Loomulik vahend selleks on funktsioonid, mille argumenditüüp on korrutistüüp.

Näiteks kahe reaalarvulise argumendiga funktsiooni $f(x, y) = \sin x + \cos y$ saab kujutada funktsioonina, mille argumenditüüp on Double $\times$ Double ja väärtusetüüp Double; siis funktsioon ise on tüüpi Double $\times$ Double $\to$ Double.

Teise võimaluse pakuvad karritatud funktsioonid.

Kui f on funktsioon, mis võtab argumendiks mingeid järjendeid, siis saab teda **karritada**; tulemuseks on funktsioon curry f, mis võtab ükshaaval samasugused argumendid nagu f ühes järjendis koos ja annab neil sama tulemuse mis f. Kui f tüüp on $A_1 \times \ldots \times A_l \to B$, siis curry f tüüp on $A_1 \to \ldots \to A_l \to B$.

Ülal vaatlesime funktsiooni f, mis võtab argumendiks ujukomaarvude paari $(x\,, y)$ ja annab sellel tulemuseks ujukomaarvu $\sin x + \cos y$. Tema karritamisel saadav funktsioon curry f on tüüpi Double $\to$ Double $\to$ Double; ta võtab argumendiks ühe ujukomaarvu x ja annab tulemuseks funktsiooni, mis omakorda võtab argumendiks ujukomaarvu y ning annab tulemuseks ujukomaarvu $\sin x + \cos y$.

Karritamisel saadavaid funktsioone nimetame **karritatud** funktsioonideks. Karritatud funktsioon on seega sama mis funktsioon, mille väärtusetüüp on funktsioonitüüp. Lugeja võis märgata, et ka curry on ise karritatud funktsioon.

Eelmises näites oli tema tüüp

$$(\text{Double} \times \text{Double} \to \text{Double}) \to \text{Double} \to \text{Double} \to \text{Double},$$

sest ta võttis argumendiks funktsiooni tüübist Double $\times$ Double $\to$ Double ja andis tulemuseks funktsiooni tüübist Double $\to$ Double $\to$ Double.

Haskellis on karritatud funktsioonid väga tavalised, enamik standardsetest mitmeparameetrilistest funktsionaalsetest suhetest (mh kõik aritmeetilised tehted) on realiseeritud karritatult.

3.3. Kõrgemat järku funktsioonid. Haskellis (ja üldse funktsionaalses programmeerimises) on väga tavalised ka funktsioonid, mille argumenditüüp on funktsioonitüüp või koguni mõni struktuuritüüp, mille komponenditüüp on funktsioonitüüp. Selliseid funktsioone nimetatakse **kõrgemat järku** funktsioonideks.

Funktsiooni **järgu** mõistet võib ka täpsustada induktiivselt funktsioonitüübi ehituse järgi.

- Funktsioone mitte sisaldava objekti järguks loetakse 0.
- Alati, kui funktsiooni argumendid saavad sisaldada kuni n. järku funktsioone ja väärtused kuni m. järku funktsioone, loetakse selle funktsiooni enda järguks $\max(n + 1, m)$.

Selle järgi on funktsioon kõrgemat järku, kui tema järk on vähemalt 2.

Näiteks naturaallogaritm tüübist Double $\to$ Double on 1. järku funktsioon, sest ujukomaarvud ei sisalda funktsioone. Ülal vaadeldud funktsioon $f(x, y) = \sin x + \cos y$ on 1. järku, sest ei andmed tüübist Double $\times$ Double ega tüübist Double ei sisalda funktsioone. Ka curry f on 1. järku; curry ise on siin aga 2. järku funktsioon, sest tema argumenditüüp on 1. järku funktsioonide tüüp.

3.4. Agarad ja laisad funktsioonid. Öeldakse, et funktsioon f on **agar**, kui $f \perp = \perp$. Vastasel korral ütleme, et f on **laisk**.

Agara funktsiooni mõiste on agara väärtustamise peegeldus matemaatilistel funktsioonidel. Agara väärtustamise korral on funktsioonid agarad, sest kõigepealt väärtustatakse argument ja kui see on bottom, st väärtustamisel tekib ületamatu viga, on ühtlasi ka funktsiooni väärtuse arvutamisel tekkinud ületamatu viga, st funktsiooni väärtus on ka paratamatult bottom. Nagu võib arvata, tuleb Haskellis tihti ette funktsioone, mis pole agarad.

3.5. Ekstensionaalsus. Funktsionaalse paradigma omadused tagavad, et Haskelli funktsioonide mõistmine abstraktse eeskirjana argumentide hulgast väärtuste hulka, nagu funktsioone matemaatikas mõistetakse, on täiesti ohutu — kui bottomid kõrvale jätta.

Bottomitega on lood sarnased andmestruktuuride bottomitega. Igas funktsioonitüübis on sees bottom, mis matemaatilises mõttes ei ole funktsioon. Suvalisele argumendile rakendades annab selline väärtuseks bottomi. Matemaatikas kehtib **ekstensionaalsus** — alati, kui funktsioonid f ja g on sama määramispiirkonnaga ja töötavad igal argumendil ühtmoodi, siis f ja g on võrdsed —, kuid Haskellis on bottom eristatav funktsioonist, mis ise ei ole bottom, kuid annab igal argumendil tulemuseks bottomi.

4. Protseduuritüübid. Reaalses elus on enamasti tarvis, et programm saaks oma töö jooksul keskkonnast uut infot lugeda, välistele sündmustele reageerida. Et kõrvalefektid on keelatud, on pidevalt muutuva keskkonna info arvessevõtmine mõneti keeruline. Funktsioon, mis seda kirjeldaks, peaks võtma keskkonna lisaks oma muudele parameetritele argumendiks. See oleks programmeerijale tülikas ja nõuaks ka võimalust keskkonna olekut keele väljendusvahenditega esitada.

Haskellis on mindud teist teed ja loodud eraldi tüübid, millesse kuuluvad just **protseduurid**, mis võivad välisest keskkonnast sõltuda ja keskkonda mõjutada. (Need tüübid võivad olla teatud erilised funktsioonitüübid, kuid see on jäetud realisatsiooniküsimuseks ja programmeerija ei pea sellest midagi teadma.)

Põhimõtteliselt on võimalik kõik algoritmid realiseerida protseduuride kaudu, aga selline lähenemine on funktsionaalses keeles halb stiil. Funktsionaalse keele üks eesmärke on eraldada matemaatilise funktsioonina esitatavad seosed kõigist ülejäänutest, muuta kood sellisel viisil ülevaatlikumaks. Seepärast tuleks funktsionaalses keeles programmeerides kasutada protseduure ainult interaktiivsete protsesside programmeerimiseks ja kodeerida niipalju andmetega manipuleerimisi kui võimalik funktsioonidega.

4.1. Edastamine. Iga tüübi A jaoks on olemas protseduuritüüp IO A, kusjuures A on protseduuri poolt **edastatava** andme ehk **edastusväärtuse** tüüp. Mudel on nimelt selline, et protseduur saab kasutada varasemate protseduuride edastatud infot, samuti lugeda seda keskkonnast juurde, ning edastab lõpuks ise mingi info, mida omakorda järgnevad protseduurid saavad kasutada.

Näiteks protseduur, mis loeb standardsisendist ühe sümboli ja edastab selle, on tüüpi IO Char. Protseduur, mis loeb standardsisendist sümboli, aga edastab tema koodi, on tüüpi IO Int.

Sama protseduur võib erinevatel kordadel sõltuvalt keskkonnast loetud infole teha erinevaid asju ja edastada erinevaid andmeid (aga edastusväärtuse tüüp ei muutu). See tähendab, et protseduur on abstraktsem kui lihtsalt mingi konkreetne vaadeldav tegevus.

Näiteks sama protseduur, mille ülesanne on lugeda standardsisendist sümbol ja see edastada, ootab erinevatel kordadel erineva ajavahemiku (kuni kasutaja sümboli sisestab) ja võib erinevatel kordadel edastada erineva sümboli (kasutaja valitu).

Protseduurid, mis midagi sisulist ei edasta, peavad vormitäiteks siiski millegi edastama; Haskelli standardteegis edastavad sellised protseduurid tüüpiliselt ühiktüübi ainsa andme ().

Kogu arvutust võib võrrelda konveieriga, kus protseduurid teevad mingeid operatsioone ja edastavad üksteisele järjest andmeid. On tagatud, et keskkonna info ei pääse sellelt konveierilt maha kõrvalefekti tekitama.

4.2. Erindid. Nagu tänapäeva programmeerimiskeeltes tüüpiline, on ka Haskellis olemas erindid — spetsiaalsed andmed erandolukordade tähistamiseks. Nad kuuluvad eraldi erinditüüpi. Erindite kasutamine võimaldab koodis tavajuhtude ja erandjuhtude töötlused üksteisest lahutada.

Üldreegel, et protseduur tüüpi IO A edastab andme tüübist A, peab paika ainult eeldusel, et protseduuri täitmine kulgeb normaalselt. Keskkonnaga suhtlemisel võib kergesti ette tulla ootamatuid olukordi, mispuhul protseduuri edasine täitmine muutub mõttetuks ja oodatud andme edastamine võimatuks. Siis protseduur edastamise asemel hoopiski heidab erindi.

5. Tüübi ühesus. Võib tekkida küsimus, kas mõni anne võib kuuluda ka mitmesse tüüpi, ehk teisiti väljendudes, kas tüüpidel saab olla mittetühi ühisosa. Vastus on eitav: Haskellis on tüübid täiesti lõikumatud. Isegi bottom on igal tüübil oma.

Niisiis tüüpidesse Int ja Integer kuuluvad täisarvud on üksteisest erinevad koopiad, samamoodi on üksteisest erinevad tüüpide Float ja Double ujukomaarvud.

Muidugi erinevad omavahel Just a ja a, kuid ka Nothing tüüpidest Maybe A on kõikide tüüpide A korral unikaalne.

1.2.2 Tüübid ja liigid

1. Tüübid laiemalt. Eelmises jaotises nähtud tüübindus toob peale andmeid klassifitseerivate tüüpide kaasa ka tüübifunktsioone — funktsioone, mis võtavad argumendiks tüüpe ja annavad ka tüüpe välja. Sellised on põhimõtteliselt erinevad kõigist funktsioonidest funktsioonitüüpides, sest nood võtavad argumendiks ja annavad tulemuseks andmeid.

Näiteks nägime, et iga tüübi A korral on olemas listitüüp List A — see tähendab, et List on funktsioon, mis igale tüübile A seab vastavusse tüübi List A. Kuna ta teisendab tüüpe tüüpideks, on ta tüübifunktsioon.

Analoogselt näiteks IO on tüübifunktsioon, mis igale tüübile A seab vastavusse tüübi IO A.

On võimalikud ka keerulisemad tüübifunktsioonid: näiteks summatüüpe konstrueeriv Either seab igale tüübile A vastavusse tüübifunktsiooni, mis igale tüübile B seab vastavusse summatüübi Either A B. Tüübifunktsioon Either on karritatud. Mõnes rakenduses on mõttekad isegi kõrgemat järku tüübifunktsioonid.

Kui jutuks on korraga tüübid ja tüübifunktsioonid, öeldakse tavaliselt mõlema kohta tüüp. See annab tüübi mõistele teise, endisest laiema tähenduse (nii nagu funktsionaalne paradigma annab andme mõistele tavalisest laiema tähenduse, haarates kaasa ka funktsioonid ja protseduurid).

Selle järgi on ka List, Maybe, IO ja Either tüübid, kuigi nad andmeid ei sisalda, vaid on hoopis teistlaadi objektid.

Siiani kasutasime me tüübi mõistet vaid kitsamas, andmehulga tähenduses; edaspidi võib juhtuda nii või teisiti, mõista tuleb sõltuvalt kontekstist.

2. Liigid. Laiemalt mõistetud tüüpidel on oma klassifikatsioon, mille annavad liigid. Tüübid, mis kujutavad endast andmekogumit, st tüübid kitsamas tähenduses, on kõik üht ja sama liiki $*$. Funktsioon, mis võtab argumendiks tüüpe, mille liik on K, ja annab väärtuseks tüüpe, mille liik on L, on ise liiki $K \to L$.

Seega näiteks List, Maybe ja IO on liiki $* \to *$, kuid Either on liiki $* \to * \to *$.

Juhime tähelepanu sellele, et tüübifunktsioonid ja funktsioonitüübid on täiesti erinevad asjad. Funktsioonitüüp on tüüp, mis koosneb andmetel töötavatest funktsioonidest; tema liik on $*$. Seevastu tüübifunktsioon on funkt-

sioon, mis töötab tüüpidel; tema liik on kõike muud kui $*$.

Kokkuvõtvalt võib nentida, et kui tüübid on pikemalt väljendudes andmetüübid, siis liigid on tüübitüübid.

1.2.3 Tüübid ja klassid

On veel teinegi tüüpe klassifitseeriv süsteem — **klassid**. Klassi kuuluvat tüüpi nimetatakse **klassi esindajaks**. Ühe klassi kõik esindajad peavad olema sama liiki, seega klassid täpsustavad liikide antavat klassifikatsiooni.

Erinevalt tüüpidest ja liikidest võivad klassid omada mittetühja ühisosa. Üks klass võib lausa tervikuna teise sees sisalduda, sellisel juhul nimetatakse esimest teise **alamklassiks** ja teist esimese **ülemklassiks**. Ühel klassil võib olla mitu alam- või ülemklassi.

Tüübiklassindus sarnaneb äratuntavalt objektorienteeritud programmeerimise klassisüsteemiga. Oluline vahe on selles, et Haskellis kuuluvad klassidesse tüübid, objektorienteeritud programmeerimises aga objektid, mis üldjuhul on tavalised andmed. Objektorienteeritud programmeerimises täidavad klassid tüüpide aset, Haskellis asuvad klassid tase kõrgemal.

Võrreldes liikidega, mis klassifitseerivad tüüpe nende olemuse järgi, on klassid suhteliselt meelevaldsed tüüpide hulgad. Klassikuuluvus tähendab täpsemalt teatavate muutujate defineeritust ja see sõltub üldiselt programmeerija suvast.

1. Eeldefineeritud klassid. Näiteks on Haskellis eeldefineeritud klass Eq, mille esindajaks on sellised tüübid, milles saab kontrollida andmete võrdumist ja mittevõrdumist. Sellesse klassi kuuluvad näiteks kõik arvutüübid, sümbolitüüp, tõeväärtusetüüp, suurusvahekorratüüp, tüübid List A ja Maybe A tingimusel, et ka A kuulub sinna, ühiktüüp, korrutis- ja summatüübid tingimusel, et komponenditüübid kuuluvad sinna, jpt. Funktsioonitüübid ega protseduuritüübid klassi Eq vaikimisi ei kuulu.

Klassi Eq alamklassi Ord esindajaks on sellised tüübid, kus lisaks andmete võrdumisele ja mittevõrdumisele on võimalik kontrollida nende omavahelist suurusvahekorda. Klassi Ord kuulub enamik arvutüüpe, sümbolitüüp, tõeväärtusetüüp, suurusvahekorratüüp, tüübid List A ja Maybe A tingimusel, et ka A kuulub sinna, ühiktüüp, korrutis- ja summatüübid tingimusel, et komponenditüübid kuuluvad sinna, jpt. Sümbolite järjestus on defineeritud kooditabeli järgi; tõeväärtusetüübil loetakse False väiksemaks kui True;

suurusvahekorrad on suurenemise järjestuses LT, EQ, GT; listi- ja korrutistüüpidel on järjestus leksikograafiline. Funktsioonid ega protseduurid sellesse klassi vaikimisi ei kuulu, kuna see nõuaks kuulumist klassi Eq.

Klassi Show esindajaks on tüübid, millesse kuuluvatel andmetel on olemas sõnekuju. Klassis Read aga on tüübid, millesse kuuluvaid andmeid on võimalik sõnekujust tuvastada. Enamik üldkasutatavaid tüüpe peale funktsiooni- ja protseduuritüüpide on mõlema klassi esindajad.

Klass Num koondab arvutüüpe. Sinna kuulumise kriteeriumiks on tüübil defineeritud aritmeetilised operatsioonid. Sellel klassil on omakorda palju alamklasse (näiteks Integral, Fractional, Floating, mille nimed räägivad iseenda eest), ise aga on ta klasside Eq ja Show alamklass.

Klassi Enum esindajaks on tüübid, millesse kuuluvate andmete puhul saab rääkida järgmise ja eelmise leidmisest. Tüüpiliselt kuuluvad siia loetelutüübid. Klassis Bounded on sellised tüübid, mille andmete seas on olemas vähim ja suurim. Tuttavatest tüüpidest kuuluvad siia ühiktüüp, Int, Char, Bool ja Ordering. Klassi Random esindajaks on tüübid, millest on võimalik juhuarvude generaatoritega juhuslikult andmeid valida. Siia kuuluvad vaikimisi olemasolevad täis- ja ujukomaarvutüübid, Char ja Bool.

2. Klassid laiendustes. Haskelli laiendustes, mis on mõnes kompilaatoris ka realiseeritud, kujutavad klassid endast üldisemalt relatsioone tüüpidel, tüübirelatsioone. Tüübirelatsioone võib käsitleda kui tüübijärjendite klasse. Kui relatsioon on unaarne, on tegemist standardse klassiga, binaarse relatsiooni puhul tüübipaaride klassiga jne, osaliste arv pole piiratud.

Analoogselt sellega, kuidas liikide sissetoomisel laiendasime tüübi mõistet tüübifunktsioonidele, võib vajadusel tüübi mõistet kasutada ka tüübijärjendite kohta. Näiteks tüübipaar võiks olla tüüp, mille liik on $* \times *$ või üldisemalt $K \times L$ mingite liikide K, L jaoks jne. (Pangem jällegi tähele, et tüübipaar on hoopis midagi muud kui paaritüüp — viimase liik on $*$.) Seega tegelikult on ka laiendatud klasside kohta mõttekas öelda, et nad klassifitseerivad tüüpe ja täpsustavad liikide antavat klassifikatsiooni.

2 Esmatutvus Haskelliga

2.1 Ettevalmistus tööks

Selles jaotises anname ülevaate levinuimatest Haskellis programmeerimise töövahenditest ja tutvustame kõige üldisemal tasemel programmide ülesehituse põhimõtteid. Käsitlus jääb siin võrdlemisi pealiskaudseks. Lugejal on soovitav mõningase vilumuse tekkimisel oma lemmikvahendite kohta nende kodulehelt või manuaalist täiendavat teavet otsida. Haskelli ennast puudutavat informatsiooni leiab hulgi Haskelli kodulehelt

`http://www.haskell.org/`.

2.1.1 Haskelli töövahendid

1. Töövahendite liigid. Haskellis programmeerimiseks on loodud nii kompilaatoreid kui interpretaatoreid. Nagu ikka, on programmi jooksutamine interpretaatoriga suurusjärke aeglasem kui sama programmi kompileerimisel saadud masinakoodi jooksutamine.

Lisaks on olemas interaktiivsed keskkonnad, mis on tõhusaks abiks lihtsamate programmide kiirel koostamisel ja jooksutamisel, muuhulgas Haskelli õppimisel. Interaktiivse keskkonna mõte on võimaldada koodi testimist "käigu pealt" ilma, et peaks selleks iseseisvalt käivitatava programmi looma. Kasutaja saab vahetult testida ühe või teise koodiosa tööd.

Täpsemalt, kasutaja sisestab interaktiivse keskkonna käsurealt avaldisi ja süsteem väärtustab neid. Kui avaldise väärtus on protseduur, siis süsteem täidab selle protseduuri. Muul juhul, kui muidugi vigu ei teki, vastab süsteem avaldise väärtusega, mille ta väljastamiseks mingil moel sõneks teeb.

Koodis võib esineda erineva iseloomuga vigu. Mõned, peamiselt süntaksi- ja tüübivead, avastab interaktiivne keskkond väärtustamist alustamata; neid nimetatakse staatilisteks vigadeks. Koodifailis esinevad staatilised vead avastab iga Haskelli realisatsioon juba faili sisselugemisel ja sellise koodifaili kasutamine pole enne nende vigade parandamist võimalik. On ka vigu, mis ilmnevad alles koodi jooksutamise käigus, need on täitmisaegsed vead.

Kõigi Haskelli realisatsioonidega tuleb kaasa süsteemne moodulite teek, kus on palju asju eeldefineeritud. Interaktiivses keskkonnas mooduli kasutamiseks peab moodul olema sisse loetud. Teegis on kesksel kohal moodul Prelude, mis sisaldab põhilisimat vajalikku, muuhulgas aritmeetikat; see moodul loetakse interaktiivse keskkonna käivitumisel sisse automaatselt.

Kõigil järgnevalt käsitletavail töövahendeil on olemas vähemalt Windowsi ja Unixi versioonid ja neid saab veebist tasuta alla laadida. Käesolevas õppevahendis eeldame Unixi-laadset operatsioonisüsteemi.

2. Hugs. Üks standardsemaid töövahendeid lihtsate Haskell-programmide koostamiseks, eriti keele õppimisel, on interaktiivne interpretaator Hugs. Hugsi koduleht asub aadressil

```
http://www.haskell.org/hugs/.
```

2.1. Hugsi kasutamine arvutamiseks. Terminaliakna käsurealt käivitub Hugs käsuga `hugs`. Seejärel on ta kohe valmis oma käsurealt vastu võtma Haskell-avaldisi ja neile reageerima. Kuna Haskelli süntaks on matemaatikalähedane, on korrektsete avaldiste sisestamine ja niiviisi interaktiivse keskkonna kasutamine kalkulaatori eest täiesti võimalik ka inimesel, kes Haskelliga kunagi varem kokku pole puutunud.

Ülesandeid

1. Käivitada Hugs.
2. Lasta Hugsil väärtustada lihtsaid Haskell-avaldisi, nt mõni aritmeetiline avaldis nagu `2 + (-3)`, `pi` vms.

2.2. Hugsi käsud. Hugsi käsurealt saab lisaks Haskell-avaldistele anda ka Hugsi käske, millest peamised on kirjeldatud joonisel 1. Hugsi käsku eristab Haskell-avaldisest alguskoolon.

Käsk	Argument	Tegevus
`:q`		väljumine
`:l`	faili/mooduli nimi	mooduli sisselugemine
`:r`		viimase sisselugemiskäsu kordamine
`:t`	avaldis	avaldise tüübi kuvamine
`:i`	nimede loend	info nimede kohta
`:b`	moodulite loend	info välja nähtavate nimede kohta
`:e`	faili nimi	tekstiredaktori avamine
`:?`		Hugsi käskude nimekirja kuvamine
`:s`		seadete nimekirja kuvamine
	muutuse tähistus	seadete muutmine

Joonis 1: Peamised Hugsi käsud.

Käsk `:r` on kasulik, kui käib ühe ja sama programmifaili korduv muutmine. Siis igal järgneval korral loetakse käsuga `:r` samast failist värske versioon sisse, ilma et kasutaja peaks faili nime pidevalt kordama.

Kuigi tekstiredaktorit ei pea muidugi Hugsist käivitama, oma failid võib teha valmis ükskõik milliste vahenditega, on Hugsi käsk `:e` hea selle poolest, et redaktorist väljumisel loeb Hugs automaatselt faili uuesti sisse.

Lisaks käsule `:s` saab Hugsi parameetrite vaikeväärtusi sättida ka keskkonnamuutujaga `HUGSFLAGS`. Näiteks võib sellele väärtuseks omistada teksti

```
-Eemacs -h32M +kls.
```

Siin `-Eemacs` seab käsu `:e` poolt käivitatavaks tekstiredakoriks Emacsi, `-h32M` reserveerib arvutusteks kasutatava mälu suurusega 32 megabaiti ning `+kls` muudab teatavad kolm seadet, mille tähendust võib uurida Hugsis, andes käsu `:s` ilma argumentideta.

Ülesandeid

3. Lasta Hugsil kuvada lihtsate avaldiste tüüpe.

4. Lasta Hugsil anda infot mõne teile tuntud identifikaatori kohta.

5. Lasta Hugsil loetleda mooduli Prelude eksporditavad nimed.

6. Lahkuda Hugsist.

3. GHC. Suuremate programmide tegemiseks, eriti kui loodava tarkvara töökiirus on oluline, on interpretaatori asemel parem kasutada kompilaatoreid. Levinuim ja arenenuim Haskelli kompilaator maailmas on Glasgow ülikoolis arendatav GHC (lühend sõnadest "*Glasgow Haskell Compiler*"). Tema koduleht asub aadressil

`http://www.haskell.org/ghc/`.

3.1. GHC interaktiivne keskkond. GHC-ga on kaasas Hugsiga sarnane interaktiivne keskkond, mis kannab nime GHCi (sõnadest "GHC *interactive*"). Hoolimata nimes sisalduvast C-tähest, pole GHCi ise kompilaator ega kasutagi oma töös kompileerimist, sest standardmoodulite korral kasutab ta varem GHC-ga valmis kompileeritud masinakoodi ning kasutaja omadefineeritud koodi ta interpreteerib.

GHCi käivitub terminaliakna käsurealt käsuga `ghci`. Interaktiivse keskkonna käsud `:q`, `:t`, `:i`, `:b`, `:?` on sarnased Hugsiga. GHC teegi moodulite sisselugemisel tuleb kasutada käsku `:m`, oma failide puhul aga `:l` ja `:r` nagu Hugsis. Käsu `:s` asemel tuleb kasutada `:set`.

Ülesandeid

7. Proovida kätt ka GHCi peal, lastes väärtustada mõned avaldised, andes mõned käsud ja lahkudes süsteemist.

3.2. Kompileerimine GHC-ga. GHC võimaldab Haskell-koodi kompileerida masinakoodiks, mida saab iseseisvalt käivitada. Selleks peab kõigepealt tegema koodifaili, milles defineeritakse käivitatav protseduur muutuja `main` väärtuseks.

Protseduuri kirjeldamisel tuleb arvestada, et millegi ekraanile toomiseks tuleb talle ette kirjutada `print`.

Teeme näiteks programmi, mis kirjutab ekraanile avaldise `2 + (-3)` väärtuse ja mis asub failis nimega `Hello.hs`. Et selle avaldise väärtus on arv, mitte protseduur, siis tuleb ette lisada `print`. Sellega on kõik soovitud operatsioonid kirjeldatud ning esimene Haskellis kirjutatud programm oleks

```
main
  = print (2 + (-3))
```
(1)

Failis nimega p asuva programmi kompileerimiseks GHC-ga masinakoodiks võib anda käsu kujul

```
ghc --make -o q p.
```

See tekitab masinakoodis programmi nimega q. Lõik "`-o` q " võib käsus ka ära jätta, kuid siis leiutab GHC ise väljundfailile mingi nime, mis ei pruugi kasutajale meeldida.

Ülesandeid

8. Tehes läbi ülalkirjeldatud sammud, tekitada GHC-ga programm `Hello`. Käivitada see ja veenduda, et programm töötab oodatud viisil.

9. Lugeda fail `Hello.hs` sisse interaktiivses keskkonnas ja käivitada sama protseduur interaktiivse keskkonna käsurealt.

4. Teised vahendid. GHC kõrval on üsna levinud ka kompilaator NHC. Tema koduleht on aadressil

```
http://www.haskell.org/nhc98/.
```

NHC töötab ise kiiremini kui GHC, kuid tema produtseeritud masinakood on GHC tekitatust aeglasem.

NHC kutsumiseks terminaliakna käsurealt on käsk `nhc98`. GHC-ga vajalik käsureaargument `--make` jääb ära, nii et koodifaili p kompileerimiseks ja tulemuse kirjutamiseks faili q sobib käsk

```
nhc98 -o q p.
```

NHC-ga tuleb kaasa kompileerimishaldur hmake koos interaktiivse keskkonnaga hi. Need programmid võimaldavad ühtsel viisil kasutada suvalist kompilaatorit. Töötamine interaktiivses keskkonnas on sarnane töötamisele Hugsis või GHCi-s, aga süsteemis on põhimõtteline erinevus: kogu kood, kaasa arvatud kasutaja sisestatavad avaldised, kompileeritakse enne käivitamist masinakoodiks. Sellele kulub tuntav ajavahemik; kui aga avaldise väärtustamine on mahukas, korvab kompileeritud koodi kiirus selle kaotuse kuhjaga.

Viimastel aastatel on ilmunud uusi Haskelli kompilaatoreid nagu seeni pärast vihma. Huvilistel on võimalik nendega tutvuda Haskelli kodulehel.

2.1.2 Koodifaili struktuur

1. Koodi paigutus. Haskell võimaldab koodi kirjutada ilma selliste tülikate eraldajateta nagu on semikoolonid paljudes keeltes, aga selleks peavad koodi osad olema failis üksteise suhtes õigesti trepitud. Õige treppimine ei ole ühene, koodi on lubatud paigutada väga erinevate maitsete järgi.

Näiteks esimene Haskell-programm oli ülal antud kaherealisena kujul (1), kuid sama programmi võib kirjutada ka kujul

```
main = print (2 + (-3)).
```

Kui kasutada kaherealist varianti, peab teine rida olema esimese suhtes positiivse taandega, muidu loetaks kaks rida eraldi deklaratsioonideks. Näites (1) on taane +2 tühikut, mis pikas perspektiivis ei ole liiga väike, kuna Haskell-kood kipub olema paljutasemelise struktuuriga.

Käesolevas õpikus on järgitud üht võimalikku paigutussüsteemi, mida täpselt jäljendades ei tohiks treppimisvigu tekkida. Igaüks võib leida katseliselt omale meelepäraseima treppimisviisi, mida ka Haskell aktsepteerib.

Seejuures tasub jälgida, et taanete tekitamisel ei kasutataks tabulaatorit, sest tabulaatori ulatus pole üheselt määratud, mistõttu võib juhtuda, et kood, mis tekstitöötlusprogrammi aknas näib olevat õigesti trepitud, ei ole seda Haskelli programmeerimiskeskkonna jaoks. Samuti võib tabulaatorite abil trepitud töötav programm üleviimisel teise keskkonda lakata töötamast.

Haskell-koodi on siiski võimalik struktureerida ka semikooloni abil. Ilma semikooloniteta koodi lugemisel Haskell tegelikult lisab sinna semikoolonid automaatselt. See kajastub tihti veateadetes, mis viitavad ootamatule semikoolonile reas, kus pole ühtki semikoolonit. Sellisel juhul tuleb aru saada, et viga on treppimises.

2. Kommentaarid. Haskell-koodi saab kahel viisil lisada kommentaare — suvalist teksti, mille süsteem oskab koodi lugemisel vahele jätta.

Kõigepealt, pikkusest sõltumata on võimalik kommentaare paigutada kommentaarisulgude `{-` ja `-}` vahele. Selline kommentaar võib omakorda sisaldada teisi kommentaare.

Lühemad kommentaarid võib aga panna kahekordse sidekriipsu järele, sellise kommentaari lõpetab esimene järgnev reavahetus. Üldjuhul on vaja kriipsude ja kommentaari alguse vahele eraldavat tühisümbolit, kuid kui kommentaar algab tähega, siis võib teda alustada otse sidekriipsude järelt.

Näiteks võib meie esimesele Haskell-programmile (1) lisada kommentaare, nii et saame teksti

```
main -- See kommentaar ulatub rea lõpuni.
  = print (2 + (-3)) -- Esimene Haskell-programm!
{-

      Pikem kommentaar.
   {- Sisemine kommentaar. -}
Pikem kommentaar jätkub. -} -- Pikema kommentaari lõpp.
```

sisaldava faili.

Ülesandeid

10. Lisada oma Haskell-faili `Hello.hs` erinevatesse kohtadesse mõlemat sorti kommentaare.

3. Standardsed failiformaadid. Haskell-koodifailide standardsed nimelaiendid on `.hs` ja `.lhs`. Nende laienditega failide lugemisel Haskelli töövahenditega pole laiendit vaja kirjutada. (Kui sama failinime põhiosa jaoks eksisteerivad nii `.hs`- kui ka `.lhs`-versioonid, siis loetakse `.hs`-fail.)

Seejuures on `.hs`- ja `.lhs`-failide oodatav formaat erinev. Kui laiendi `.hs` puhul loetakse oluliseks kogu faili sisu, siis laiendi `.lhs` puhul loetakse ainult sümboliga `>` algavaid ridu, kust ka see algussümbol jäetakse kõrvale. Loetav sisu moodustub niisiis sümboliga `>` algavate ridade algussümbolile järgnevaist osadest, sellele rakenduvad muuhulgas endised treppimise ja kommentaaride kirjutamise reeglid. Mitteloetavatel ridadel võib programmeerija kirjutada lisakommentaare või hoida millist tahes infot.

Lisanduv l-täht laiendis tuleb ingliskeelsest sõnast ‘*literate*’.

Ülesandeid

11. Teha fail `Hello.lhs`, milles on sama programm mis ülalkirjeldatud failis `Hello.hs` koos samade kommentaaridega. Lisada kommentaare koodivälistele ridadele.

2.1.3 Moodulid

1. Standardteek. Hetkel kehtiv Haskelli standard Haskell 98 määrab 17 moodulit. Kõik need sisalduvad kõigi meie käsitletud Haskelli realisatsioonide teekides.

Standardteek pole just suur, kuid ta moodustab tänapäeval vaid pisikese osa kogu olemasolevast moodulite hierarhiast. Viimane sarnaneb Java klassi- ja paketihierarhiaga, kus klassi täisnimi kujutab endast aadressi, milles esinevad järjest hierarhia üksused kõrgemast madalama suunas, üksteisest punktiga eraldatud.

Näiteks Haskelli standardteegi moodul, millest saab sümbolitega seonduvaid operatsioone, on nimega Data . Char. Standardmoodulid on kättesaadavad ka ilma täisteeta, nii et moodulile Data . Char saab viidata ka palja nimega `Char`.

2. Moodulite tähtsus. Moodulisüsteemil võib näha kahte rolli. Mõlemas on Haskelli moodul Java klassi analoog.

2.1. Kasutaja aspekt. Teegi kasutaja aspektist teenivad moodulid objektide temaatilise jagamise eesmärki.

Nagu öeldud, saab sümbolitega seonduvaid operatsioone moodulist Data . Char. Sarnaselt saab operatsioone ratsionaalarvudega moodulist Data . Ratio, operatsioone kompleksarvudega moodulist Data . Complex, failioperatsioone moodulitest System . IO ja System . Directory jne. Need kõik on standardmoodulid.

Vaatame lähemalt moodulit Data . Ratio. See võimaldab teha täpseid arvutusi harilike murdudega vastanduvalt ujukomaarvudele, millega opereerimine on ebatäpne. Murrujoone aset täidab protsendimärk %, mida pole teegi kujundamisel loetud põhiliste operaatorite hulka ja mida seetõttu moodulist Prelude ei saa.

Moodul Data . Ratio interaktiivses keskkonnas sisse loetud, saame anda arvutile ülesandeks teha tehteid nagu näiteks `1 % 2 - 1 % 6`, millele saame täpse vastuse `1 % 3` (ratsionaalarvude lõppkuju, millele nad väärtustamisel viiakse, on esitus taandumatu murruna).

Ülesandeid

12. Lugeda interaktiivses keskkonnas sisse moodul Data . Ratio ja sooritada mõned tehted harilike murdudega.

2.2. Kirjutaja aspekt. Teegi looja aspektist teenivad moodulid eelkõige koodi sobiva jaotamise eesmärki, mis on eelmisest mõneti erinev.

Mitte kõik, mis teatavas moodulis defineeritakse, ei pea jääma selle mooduli kaudu kasutajale kättesaadavaks. Väljaspoolt jäävad nähtavaks ainult nimed, mida moodul ekspordib.

Teisest küljest ei pea kõik, mis mooduli kaudu kasutajale kättesaadav on, olema just selles moodulis defineeritud. Moodul võib ka teiste moodulite

eksporditavaid nimesid importida ja omakorda edasi eksportida samadel alustel moodulis endas defineeritud nimedega.

Esmase tutvuse moodulitega nende programmeerija aspektist võib teha Hugsi teegi peal, sest Hugsi distributsioonis on teegi moodulid algteksti kujul. GHC distributsioonis on teek vaid kompileeritud kujul, algtekste pole. Algtekstid on siiski valdavalt samad, kuna pärinevad samast repositooriumist.

Keelereeglid jätavad programmeerijale koodi jaotamiseks moodulitesse üsna vabad käed. Kui objektorienteeritud keelte reeglid nõuavad sama klassi isendimeetodite paigutamist sama klassi sisse, siis Haskell mingeid taolisi kitsendusi ei sea. Haskelli moodulid võivad lausa vastastikku teineteist importida ja kumbki enda ja imporditud nimesid segiläbi eksportida.

Ühelt poolt võimaldab selline vabadus moodulisüsteemi eesmärke maksimaalselt saavutada, teisest küljest aga jätab see täielikult programmeerija vastutusele, et mooduliteks jaotus tõepoolest mõistlik saaks.

Ülesandeid

13. Visata silm peale Hugsi moodulite teegile.

14. Lugeda interaktiivses keskkonnas sisse mõni uus moodul ja küsida selle mooduli eksporditavate nimede loend.

3. Mooduli kirjutamine.

3.1. Mooduli ehitus. Moodul koosneb kas mooduli päisest ja mooduli kehast või ainult mooduli kehast. Mooduli päise lihtsaim kuju on

```
module m where
```

kus `m` on mooduli nimi. Lihtsaim mooduli keha on lihtsalt tühi, aga üldiselt mooduli keha sisaldab deklaratsioone.

Programmi moodul, mis defineerib selle muutuja `main`, mille väärtuseks on käivitatav protseduur, peab olema nimega `Main`. Kui mooduli päis puudub, siis loetaksegi mooduli nimeks `Main`.

Meie programm (1) failis `Hello.hs` oli ilma päiseta ja sisaldas ühe deklaratsiooni. Et selle mooduli nimeks loetakse `Main`, näitab ka interaktiivne keskkond faili sisselugemisel.

Eri moodulid peavad paiknema eraldi failides. Mooduli nimi võib sisaldada tähti, numbreid ja alakriipsu ning peab algama suurtähega. Failinimi ilma laiendita peab reeglina kokku langema selles failis defineeritava mooduli nimega. Hugs ning GHC ja GHCi siiski lubavad olla moodulist sõltumatu nimega failil, mida nad jooksvast kataloogist otse sisse loevad.

Tänu viimasele klauslile võis näiteprogramm (1) olla failis `Hello.hs`, ilma selleta pidanuks fail olema `Main.hs`.

Ülesandeid

15. Milline on kõige väiksem võimalik fail, mille interaktiivsed keskkonnad Haskelli moodulina sisse loevad?

3.2. Import ja eksport. Moodulis saab kasutada nimesid oma tähenduses, mis defineeritakse samas moodulis või imporditakse mõnest teisest. Keskse mooduli Prelude eksporditavad nimed imporditakse vaikimisi, teiste moodulite nimede importimiseks tuleb kirjutada impordideklaratsioon. Impordideklaratsioonid peavad asuma mooduli keha alguses enne kõiki muid deklaratsioone. Lihtsaim impordideklaratsioon on kujul

```
import m,
```

kus m on mooduli nimi.

Näiteks kui tahame kirjutada harilikke murde oma moodulis, peame seal importima mooduli Data . Ratio. Programm

```
import Data.Ratio

main                                                    (2)
  = print (1 % 2 - 1 % 6)
```

sooritab tehte $\frac{1}{2} - \frac{1}{6}$ ja kirjutab vastuse ekraanile.

Importimisel võib tekkida nimepõrge, kui mõni moodulis defineeritav nimi tuleb sisse ka impordi kaudu või imporditakse mõni nimi mitmest moodulist. Kui nime algpäritolu pole üheselt tuvastatav, on nime kasutamisel vaja tema ette punktiga eraldatult lisada moodul ehk kirjutada adresseeritud nimi. Muudel juhtudel pole seda vaja, kuid see on alati lubatud.

Näiteks võib programmis (2) kirjutada % asemel igal pool `Data.Ratio.%`.

Eksportimise kohta võib lühidalt öelda niipalju, et vaikimisi ekspordib moodul parajasti kõik selle, mis on otseselt temas defineeritud, st kõik, mis on temas kasutatav, peale imporditud nimede. Aga programmeerija saab esitada mooduli päises mooduli nime järel sulgudes **ekspordiloendi**, kus ekspordiüksused eraldatakse üksteisest komaga.

Näiteks võiksime moodulile (2) tema mõtet muutmata kirjutada algusse päise `module Main where` või ka päise `module Main (main) where`.

Kui panna päis `module Main () where`, siis see moodul midagi ei ekspordiks, isegi mitte muutujat `main`. Samas võib selle mooduli panna hoopis rohkem nimesid eksportima, lisades ekspordiloendisse moodulist Data . Ratio imporditud nimesid. Näiteks päise `module Main (main, Rational) where` korral ekspordib moodul ka ratsionaalarvutüübi nime `Rational`.

Iga eksporditav nimi eksporditakse ainult ühes tähenduses, isegi kui ta mooduli sees esineb importide tõttu mitmes. Seega võib moodulite importimisel arvestada, et ükski import ei too üksi kaasa nimepõrget.

Põhimõtteliselt saab ka impordideklaratsioonile lisada mooduli nime järele täpsustuse, millised nimed importida ja millised mitte.

Ülesandeid

16. Kirjutada moodul, mille sisselugemisel interaktiivses keskkonnas on adresseerimata kasutatavad nii ratsionaalarvud kui kompleksarvud.

17. Kirjutada moodul (2) faili. Lugeda ta seejärel interaktiivses keskkonnas sisse ja nõuda vastava käsuga tema ekspordiloendit. Proovida ka olukordi, kus moodulile on lisatud ülal kirjeldatud viisidel päis.

2.2 Lihtsad avaldised

Koodijuppi, mis vastavalt keelereeglitele kannab oma kontekstis iseseisvat tähendust, nimetame **süntaktiliseks üksuseks**. **Avaldised**, mida saab anda interaktiivsele keskkonnale väärtustamiseks, moodustavad ühe tähtsa süntaktiliste üksuste liigi, aga nad pole muidugi ainsad.

Iseseisvalt käivitatava programmi tegemisel ja moodulite kirjutamisel juba koostasime teistsuguseid süntaktilisi üksusi — deklaratsioone.

Väga üldisel tasemel eristuvad andmeavaldised, mida siiani oleme näinud ja mille väärtus on anne, ning tüübiavaldised, mille väärtus on tüüp. Kui jutt on lihtsalt avaldistest, mõeldakse enamasti andmeavaldisi. Interaktiivse interpretaatori käsurealt saab väärtustada vaid andmeavaldisi.

Pangem tähele, et kogu järgnevas on pidevalt korraga vaatluse all ühelt poolt süntaktiline maailm, teisalt matemaatiliste objektide maailm ning kolmandaks ka arvutusprotsessid. Et need maailmad peegelduvad üksteises, siis on ühtesid ja samu termineid kasutatud erinevate maailmade asukate märkimiseks. See on programmeerimisalases kirjanduses tavaline.

Näiteks oleme tuttavad funktsioonide rakendamisega matemaatilises mõttes, kuid süntaktilisi konstruktsioone, mille tähenduseks on funktsiooni rakendamine, nimetame samuti funktsiooni rakendamiseks ning seejuures saab veel vahet teha rakendamisel kui operaatoril ja rakendamisel kui sellega koostatud avaldisel. Lõpuks nimetame funktsiooni rakendamiseks ka rakendamisega konstrueeritud avaldise väärtustamisprotsessi.

Samuti näiteks on matemaatilises maailmas paarid ja süntaktilises maailmas paarid (formaalsed paarikujul kirjutised).

2.2.1 Muutujad ja konstruktorid

Haskelli lihtsaimad süntaktilised üksused on muutujad ja konstruktorid. Analoogselt avaldiste jaotamisele jaotuvad muutujad väärtuse järgi andmemuutujateks ja tüübimuutujateks, konstruktorid aga andmekonstruktoriteks ja tüübikonstruktoriteks.

1. Olemus. Muutujad ja konstruktorid vastanduvad üksteisele selle poolest, kuidas arvutusprotsessis nendega ümber käiakse.

1.1. Muutuja. Muutuja tähistab Haskell-programmi täitmisel *a priori* täiesti tundmatut objekti. Et saada mingitki infot väärtuse kohta (lisaks muidugi tüübiinfole, mis on enne teada), peab muutujat väärtustama.

Nii juhtub näiteks, kui interaktiivse keskkonna käsurealt sisestada `pi`. Kuna `pi` on muutuja, leitakse tema väärtus ja antakse päringu vastuseks.

Kui sõltumata käsutatavatest ressurssidest ja muust keskkonna seisundist ei saa avaldise väärtust normaalsel kujul leida, siis loetakse avaldise väärtuseks bottom, $\bot$. Normaalne kuju võib puududa kahel põhjusel: väärtusta-

mine lõpeb täitmisaegse veaga või ei lõpe üldse.

Näiteks muutuja `undefined` väärtustamine ei anna välja muud kui veateate; niisiis muutuja `undefined` väärtus on $\bot$. Pangem tähele, et tegemist on defineeritud muutujaga, muidu tekiks süntaksiviga, mitte täitmisaegne viga.

1.2. Konstruktor. Seevastu konstruktor tähistab Haskell-programmi täitmisel lõppväärtust, mida enam teisendada pole vaja. Konstruktor loetakse täisinformatsiooniks. Konstruktori rollis on muuhulgas mitmesugused konstandid: kõik arvkonstandid ja kõik sümbolkonstandid sõltumata üleskirjutusviisist.

Näiteks kuuluvad siia arvkonstandid `0`, `1`, `1.5` jne, tõeväärtusi märkivad `True` ja `False` ning tühja listi märkiv `[]`.

Täisarvkonstante võib üles kirjutada kümnend-, kaheksand- ja kuueteistkümnendsüsteemis (viimases kahes vastavalt prefiksiga `0o` ja `0x`), ujukomaarvkonstante ainult kümnendsüsteemis kas otse või eksponendi abil.

Sümbolkonstante kirjutatakse sümbolina apostroofide vahel. Sümbolit võib seal asendada tema langjoonega algav kood. Lubatud on C-st ja Javast tuntud koodid, aga Haskellis saab kodeerida sümboleid ka kujul `\`s, kus s on sümboli number Unicode'i tabelis. See kood võib olla esitatud nii kümnend-, kaheksand- kui ka kuueteistkümnendsüsteemis, kahel viimasel juhul vastavalt prefiksiga `o` ja `x`.

Nii on korrektsed sümbolkonstandid näiteks `'a'`, `'!'`, `'\n'` (viimane kodeerib Unixi reavahetust), samuti näiteks `'\97'` ja `'\x61'`, mis mõlemad tähendavad a-tähte.

Langjoon sümbolina tuleb alati asendada koodiga, lihtsaim võimalus on kujul `\\`. Sümbolkonstandis on vajalik ka apostroofi kodeerimine kujul `\'`.

On loomulik ja tüüpiline, et interaktiivse keskkonna käsurealt konstruktori sisestamisel tuleb vastuseks seesama asi. Mõnikord tuleb vastuseks midagi muud, sest väljundi sõneksteisendamisel võib objekt saada teistsuguse kuju kui sisestas kasutaja.

2. Nimepiirangud. Haskelli leksika seab muutuja- ja konstruktorinimedele piirangud.

Muutujanimed peavad üldjuhul koosnema tähtedest, numbritest, alakriipsust ja apostroofist, kusjuures andmemuutuja nimi peab siis algama väiketähe või alakriipsuga. Andmemuutujate puhul on lubatud ka nime koosne-

mine sümbolitest nimekirjas

```
+, -, *, /, ^, =, <, >, &, |, $, :, !, ?, ., %, \, @, ~, #,
```
(3)

mispuhul nimi ei tohi alata kooloniga.

Ka konstruktorinimed peavad üldjuhul koosnema tähtedest, numbritest, alakriipsust ja apostroofist, kuid peavad algama suurtähega. Andmekonstruktori nimi võib koosneda ka sümbolitest loetelus (3), mispuhul ta peab algama kooloniga.

Eeldefineeritud konstruktorite hulgas on rida erandlikke, mille nimi ei vasta kirjeldatud tingimustele.

Ülalvaadelduist on reeglipärased konstruktorid `True` ja `False`, ülejäänud (mh arv- ja sümbolkonstandid) on erandlikud.

Seega on võimalik muutuja ja konstruktori vahel vahet teha ka kirjapildi järgi. Võtmesõnad on enamasti keelatud nii muutuja kui konstruktorina.

Ülesandeid

18. Sisestada interaktiivse keskkonna käsurealt avaldis, mis sisaldab (a) väiketähega algavat defineerimata nime, (b) suurtähega algavat defineerimata nime. Märgata erinevust veateates. Võrrelda veateatega, mis tuleb muutuja `undefined` väärtustamisel. Mida erinevused väljendavad?

19. Millised nimedest `p`, `p1`, `p_1`, `pP`, `p_P`, `p-P`, , `P`, `P_p`, `Pp`, `_P`, `P+`, `+-+`, `:-:`, `:` sobivad muutujaks ja millised konstruktoriks?

3. Tüübitaseme muutujad ja konstruktorid. Ka esimesed kokkupuuted tüübiavaldistega võib teha interaktiivses keskkonnas, küsides andmeavaldiste tüüpe käsuga `:t`.

3.1. Tüübikonstruktorid. Põhitüübid saab väljendada tüübikonstruktoritega.

Näiteks tõeväärtusetüüpi väljendab `Bool` ja sümbolitüüpi `Char`. Lühikese ja pika täisarvu tüüpe tähistavad vastavalt `Int` ja `Integer`, väikese ja suure täpsusega ujukomaarvu tüüpe vastavalt `Float` ja `Double`.

3.2. Lokaalsed tüübimuutujad. Andmeavaldist nimetatakse monomorfseks, kui tema väärtuse tüüp on üheselt määratud, ja polümorfseks, kui tal võib väärtusi olla mitmest tüübist. Nähtust, kus sama süntaktiline üksus omandab väärtusi mitmest tüübist, nimetatakse polümorfismiks.

Näiteks aritmeetilised avaldised on enamasti polümorfsed, sest arvud võivad kuuluda paljudesse eri tüüpidesse (Int, Integer, Double jne).

Haskellis on kõik tüübid, milles konkreetne andmeavaldis omab väärtusi, alati esitatavad tüübiparameetrite abil ühtse perena, mida väljendab üks tüübiavaldis; viimast loetaksegi avaldise **tüübiks**. Tüübiparameetrite tähistamiseks polümorfse avaldise tüübis kasutatakse tüübimuutujaid. Sellised muutujad on nähtavad ainult konkreetse tüübiavaldise piires. Lokaalsete tüübimuutujate nimed peavad algama väiketähe või alakriipsuga.

Näiteks muutuja `undefined` on polümorfne, tema väärtus on $\perp$ suvalisest tüübist. See tähendab, et `undefined` tüüpi väljendab näiteks paljas tüübimuutuja `a`. See muutuja märgib üht suvalist tüüpi. Et ta on nähtav ainult selles kohas, võib ta vabalt n-ö ümber nimetada ehk kasutada tema asemel mistahes teist muutujat.

Osa polümorfsete avaldiste tüüpe nõuab tüübimuutujatele **kitsendusi** — formaalselt kirja pandud klassikuuluvustingimusi. Kitsendusi esitavat lisandit nimetatakse **tüübikontekstiks**. Üldjuhul kujutab see endast komadega eraldatud kitsenduste nimekirja ümarsulgudes. Tüübikontekst kirjutatakse tüübiavaldise algusse ja eraldatakse järgnevast tingimusteta osast märgiga `=>` .

Näiteks arvkonstandi `2` tüüp on `(Num a) => a`; see tüübiavaldis väljendab suvalist tüüpi A, mille korral A kuulub klassi Num. Järelikult `2` omab väärtust parajasti igas tüübis A klassist Num ehk, lihtsamalt öeldes, igas arvulises tüübis.

3.3. Tüübiannotatsioonid. Üks koht, kus tüübiavaldisi koodis kasutatakse, on **tüübiannotatsiooniga avaldised** kujul

$$\mathtt{a} \ \mathtt{::} \ \mathtt{t}, \tag{4}$$

kus `a` on andmeavaldis ja `t` tüübiavaldis. Pole juhus, et interaktiivsed keskkonnad annavad päringutele avaldise tüübi kohta just sellisel kujul vastuse.

Avaldise (4) tüübikorrektsuse jaoks on tarvilik, et `t` väljendaks ainult selliseid tüüpe, millest `a` omab väärtusi. Kui nii, siis on tüübiannotatsiooniga avaldise (4) tüüp parajasti `t` ning väärtus võrdne annotatsioonita avaldise `a` väärtusega (tüüpides, mida `t` väljendab). Tüübiannotatsiooni mõte on üldjuhul tüüpi kitsendada (mitte meelevaldselt teisendada, seda annotatsioon ei võimalda).

Näiteks `True :: Bool` ja `'a' :: Char` on korrektsed tüübiannotatsiooniga avaldised. Andes nad interaktiivse keskkonna käsurealt väärtustada, saame vastuseks vastavalt `True` ja `'a'` ehk samad, mis ilma annotatsioonita.

Nende avaldiste puhul on annotatsioon praktiliselt mõttetu. Vaatame aga avaldisi `2 :: Int` ja `2 :: Double`: kuna `2` omab väärtust kõigis arvutüüpides, siis annotatsioon täpsustab, millise tüübi arvu 2 on mõeldud. Võib märgata, et interaktiivses keskkonnas väärtustamisel on `2 :: Double` puhul vastus teistsuguse kujuga kui `2` ja `2 :: Int` puhul.

Et aga sümbolitüüp pole arvuline (ei kuulu klassi Num), siis `2` ei oma väärtust tüübist Char ning annotatsiooniga avaldis `2 :: Char` annab tüübivea.

Igati korrektne on tüübiannotatsiooniga avaldis `2 :: (Num a) => a`, kuid siin ei lisa annotatsioon mingit kasulikku infot. Küll aga on sisukas kirjutada `2 :: (Floating a) => a`. Selle avaldise väärtus saab olla 2 suvalisest ujukomaarvulisest tüübist.

Ülesandeid

20. Küsida interaktiivses keskkonnas muutuja `pi` tüüp ja saada sellest aru.

21. Teha katseliselt kindlaks, kas teie kasutatavas versioonis on ujukomaarvud tüüpides Float ja Double erineva või ühesuguse täpsusega.

3.4. Vaiketüübid. Kui polümorfse avaldise kontekstist ei selgu, millisesse tüüpi kuuluvat väärtust on mõeldud, siis võib süsteemil tekkida raskusi avaldise väärtustamisel.

Põhimõtteliselt tekib interaktiivses keskkonnas arvavaldiste nagu `2` sisestamisel just selline olukord, sest kui teda interpreteerida täisarvuna, siis tuleb vastus väljastada kujul `2`, kui aga ujukomaarvuna, siis kujul `2.0`.

Üldjuhul võib taolises olukorras tüübiviga tekkida. Arvutüüpide puhul aga nii siiski ei juhtu, vaid süsteem valib võimalike tüüpide seast välja ühe. Vaikimisi loetakse, et avaldised, mis saavad omada kõiki täisarvutüüpe, on tüüpi `Integer` ning avaldised, mis saavad omada kõiki ujukomaarvutüüpe, kuid mitte täisarvutüüpe, on tüüpi `Double`.

Näiteks annab interaktiivne keskkond avaldise `2` ja `2 :: (Num a) => a` väärtustamisel vastuseks `2`, sest süsteem loeb ta vaikimisi täisarvuks (täpsemalt tüüpi `Integer`). Kirjutades aga `2 :: (Floating a) => a`, tuleb vastuseks `2.0`, sest tüübiannotatsioon välistab täisarvulise interpretatsiooni (nüüd loetakse tüübiks `Double`).

2.2.2 Infiksoperaatorid

Operaatoriks nimetame muutujat või konstruktorit, mille väärtus on funktsioon.

Operaatorit nimetatakse infiksoperaatoriks, kui teda kasutatakse infiksselt, st oma argumentide vahel.

Oleme juba kätt proovinud aritmeetiliste avaldistega nagu näiteks `2 + 3` ja `1 % 2 - 1 % 6`, kus avaldised on konstrueeritud arvkonstantidest infiksoperaatorite `+`, `-`, `%` abil. Muidugi on olemas ka korrutamine `*` ja murdarvude jagamine `/` (murrujoone `%` kasutamiseks peavad argumendid olema täisarvud), näiteks on korrektne ujukomaarvutüüpi avaldis `2 * pi / 3`.

Infiksoperaatorite ring on Haskellis väga lai. Aritmeetikagi ei piirdu mainitud viie tehtemärgiga. Kui jagamisest oleme näinud kaht varianti, siis astendamiseks on olemas lausa kolm infiksoperaatorit: `^`, `^^` ja `**`. Neist esimene sobib juhul, kui astendaja on naturaalarv, teine on murdarvu tõstmiseks täisarvulisse astmesse, kolmas on ujukomaarvude astendamiseks.

Ka võrdlemiseks kasutatavad `==`, `/=`, `<=`, `<`, `>=`, `>` on infiksoperaatorid. Nende abil on võimalik kirjutada avaldisi nagu näiteks `pi == 3`, `2 + 2 /= 5`, `9 ^ (9 ^ 9) <= (9 ^ 9) ^ 9`, mis kõik on süntaktiliselt ning tüübiliselt korrektsed avaldised. Operaatorid `==` ja `/=` tähendavad vastavalt võrdust ja mittevõrdust, ülejäänute tähendus on sarnane teiste programmeerimiskeeltega.

Loogiliste avaldiste tegemiseks on kasutatavad infiksoperaatorid `&&` ja `||`, mille väärtuseks on vastavalt konjunktsioon ja disjunktsioon. Nendega moodustatud loogilise avaldise väärtustamist alustatakse vasakust argumendist. Kui `&&` vasak argument väärtustub vääraks, siis paremat argumenti ei väärtustata. Samuti kui `||` vasak argument väärtustub tõeseks, siis paremat ei väärtustata.

1. Prioriteet ja assotsiatiivsus. Igal infiksoperaatoril on olemas kaks atribuuti, prioriteet ja assotsiatiivsus, mida on vältimatult vaja arvestada infiksoperaatorite vähegi keerukamal kasutamisel. Need atribuudid määravad avaldise ülesehituse kohtades, kus sulud seda ei tee. Piltlikult öeldes nad näitavad, kui tugevalt tehtemärgid oma ümbrust seovad.

Prioriteet ja assotsiatiivsus on kõigile matemaatikast mingil määral tuttavad, kuid kuna Haskellis on infiksoperaatoritel nii suur osa, on vaja neid Haskellis programmeerimisel veelgi tähelepanelikumalt jälgida.

1.1. Prioriteet. Prioriteete märgivad Haskellis täisarvud 0-st 9-ni, suurem arv vastab kõrgemale prioriteedile. Kohtades, kus sulud jätavad avaldise struktuuri lahtiseks, seovad kõrgema prioriteediga infiksoperaatorid leksee-

me enda ümber tugevamalt kui madalama prioriteediga infiksoperaatorid.

Aritmeetiliste operaatorite prioriteetide järjekord on sama mis matemaatikas, st astendamised on kõrgema prioriteediga kui korrutamine ja jagamine, mis omakorda on kõrgema prioriteediga kui liitmine ja lahutamine. Seega näiteks avaldis `2 * 3 ^ 8 + 1` on sama mis avaldis `(2 * (3 ^ 8)) + 1`. Võrdlusoperaatorid on aritmeetilistest operaatoritest madalama prioriteediga. Loogilistest operaatoritest on && kõrgema prioriteediga kui `||` ning nad mõlemad on madalama prioriteediga isegi võrdlusoperaatoritest.

1.2. Assotsiatiivsus. Assotsiatiivsus tuleb mängu, kui sulud avaldise struktuuri ei määra ja ka prioriteet on võistlevail infiksoperaatoritel võrdne (st toimub võrdse prioriteediga infiksoperaatorite ahelrakendamine: tulemust kahel argumendil opereeritakse omakorda kolmandaga jne mingi arv kordi). Infiksoperaator saab olla vasak-, parem- või mitteassotsiatiivne.

Vasakassotsiatiivsete operaatorite puhul seovad vasakpoolsed esinemised parempoolsetest tugevamalt, paremassotsiatiivsete operaatorite puhul vastupidi.

Näiteks astendamisoperaatorid on paremassotsiatiivsed, seega avaldis `9 ^ 9 ^ 9` on võrdne avaldisega `9 ^ (9 ^ 9)`, mitte avaldisega `(9 ^ 9) ^ 9`.

Seevastu `-` ja `/` on sarnaselt matemaatikale vasakassotsiatiivsed.

Assotsiatiivsus on oluline ka nendel operaatoritel, mille puhul lõppväärtus nende sooritamise järjekorrast ei sõltu, sest süsteem peab ju mingi konkreetse järjekorra valima, milles neid teha. Nii on määratud ka `+` ja `*` vasakassotsiatiivseks.

Mitteassotsiatiivsete operatsioonide võistlevad esinemised on üldse keelatud. Nende ahelrakendamisel tuleb ahela täpne ülesehitus sulgudega näidata, vastasel korral on tegemist süntaksiveaga. Sama kehtib juhul, kui ahelas esinevad vastandliku assotsiatiivsusega infiksoperaatorid.

Vaikimisi on infiksoperaatorid vasakassotsiatiivsed prioriteediga 9. Infiksoperaatorite kohta, mille prioriteet või assotsiatiivsus erineb vaikeväärtusest, annab käsk `:i` interaktiivses keskkonnas teada lisaks tüübile ka prioriteedi ja assotsiatiivsuse.

Ülesandeid

22. Teha interaktiivse keskkonna abil kindlaks operaatori `%` prioriteet teiste käsitletud infiksoperaatorite suhtes.

23. Kirjutada interaktiivse keskkonna käsurealt avaldis, mille tüüp on `Bool` ja mille väärtustamisega kontrollitakse, kas $\pi \leqslant \frac{22}{7} < \sqrt{10}$. Kasutada võimalikult vähe sulge.

24. Arvutada interaktiivses keskkonnas arvu 1.6^{16} esitus taandumatu murruna.

2. Infikskonstruktorid. Kui infiksoperaatorid vajavad koos oma argumentidega väärtustamist, siis on nad muutujad.

Kõik seni vaadeldud infiksoperaatorid on muutujad.

Aga operaatorid saavad olla ka konstruktorid. Sellised on tüüpiliselt seotud andmestruktuuride koostamisega.

2.1. Paarikonstruktor. Mõneti lihtsaim infiksoperaator-konstruktor on paarimoodustaja `,`.

Näiteks avaldise `(2 , -3)` väärtus on paar $(2\,,-3)$. Siin asub `,` argumentide `2` ja `-3` vahel, st kasutus on infiksne. Avaldise `(1 + 1 , 2 - 5)` väärtus on samuti $(2\,,-3)$.

Kui avaldis on paarikujul, siis tema väärtustusprotsess võib teisendada paari välju (komponente), kuid sulud ja koma arvutuse käigus ära kaduda ei saa — paar jääb paariks. Seega paarimoodustamisoperaator on konstruktor.

See on ühtlasi esimene näide konstruktorist, mille väärtus on funktsioon — varasemad vaadeldud konstruktorid argumente ei võtnud.

Haskelli paarimoodustamisoperaator pole siiski reeglipärane konstruktor, vaid erandlik, sest see `,` nõuab alati konstrueeritud avaldise ümber sulge ja pealegi ei alga ta nimi kooloniga.

Ülesandeid

25. Väärtustada interaktiivses keskkonnas võimalikult lühike avaldis, mis sisaldab kaht paari.

2.2. Mittetühja listi konstruktor. Teisena tutvume listikonstruktoriga `:`, mis samuti on infikskonstruktor, seekord reeglipärane. Tema väärtus on funktsioon, mis, saades argumendiks suvalise andme x ja sama tüüpi andmete listi l, annab välja listi $x : l$ ehk listi, mille pea (esimene element) on x ja saba (ülejäänud elementide list) on l.

Näiteks 5-elemendilist listi, mille elemendid on järjest 1, 3, 5, 7, 9, väljendab avaldis `1 : (3 : (5 : (7 : (9 : []))))`. Kuid infiksoperaator `:` on paremassotsiatiivne, mistõttu võib kõik sulud ära jätta ja kirjutada lihtsalt `1 : 3 : 5 : 7 : 9 : []`.

Nagu ka paarikonstruktor, märgib `:` andmestruktuuri väljade — nimelt listi pea ja saba — eraldajat ning operaatoriga `:` moodustatud avaldisest see operaator teisendamise käigus kuhugi kaduda ega paigast nihkuda ei saa. See tähendab, et infiksoperaator `:` on konstruktor.

Interaktiivses keskkonnas listidega töötamisel võib algul segadust põhjustada asjaolu, et liste esitatakse kasutajale elementide loendina kantsulgudes. See ei tähenda, et arvutatud väärtusest need koolonid ja tühi list puuduvad, vastupidi. Lihtsalt sõnekuju listide esitamiseks on valitud teistsugune, sisemist ehitust peitev.

Algajad kipuvad mittetühja listi konstruktorit segi ajama infiksoperaatoriga `++`, mis on muutuja ja mille väärtuseks on kahe listi **konkatenatsioon**, st funktsioon, mis sama tüüpi elementidega listidel l ja k annab tulemuseks listi, mille alguses on kõik l elemendid (muutmata järjestuses) ja kui l on lõplik, siis selle järel ka kõik k elemendid (muutmata järjestuses). Konstruktorit `:` ei saa kasutada nii nagu operaatorit `++`, sest esimese argumendi tüüp on erinev.

Näiteks igati korrektne on avaldis `(1 : 2 : []) ++ (1 : 3 : 5 : [])`, väärtuseks lõplik list elementidega 1, 2, 1, 3, 5. Tüübivigased on aga näiteks `1 : 2`, samuti `1 ++ 2 ++ []` ja `(1 : 2 : []) : (1 : 3 : 5 : [])`.

Pangem veel kõrva taha, et konkatenatsiooni arvutuse käigus ehitatakse parempoolse listi ette vasakpoolse koopia, parempoolset listi ei uurita üldse. See näitab, et konkatenatsiooni arvutusaeg sõltub lineaarselt vasakpoolse listi pikkusest, kuid ei sõltu parempoolse listi pikkusest. Selle tõttu on operaator `++` tehtud paremassotsiatiivseks, sest nii on arvutus efektiivsem.

Tõepoolest, arvutades avaldist `(a ++ b) ++ c`, kopeeritakse kõigepealt vasakpoolne list keskmise listi ette ja seejärel mõlemad koos parempoolse listi ette. Esimese listi elemente tõstetakse seega ümber kaks korda. Avaldise `a ++ (b ++ c)` korral kopeeritakse vasakpoolne list ülejäänute konkateneerimise tulemuse ette, mistõttu teist korda samu elemente ümber tõsta pole vaja.

Ülesandeid

26. Sisestada interaktiivses keskkonnas mõned listid, jälgida, mida ta vastab, ja

saada aru põhimõttest.

27. Selgitada välja operaatorite `:` ja `++` prioriteet senivaadeldud infiksoperaatorite suhtes.

28. Arvutada interaktiivses keskkonnas 3-elemendiline list, mille elemendid on baitide arvud vastavalt kilobaidis, megabaidis ja gigabaidis. Kasutada võimalikult vähe sulge.

29. Arvutada interaktiivses keskkonnas 2-elemendiline list, mille esimene element ütleb, kas radiaan on väiksem kui 57°, ja teine, kas $30 < \pi^3 \leqslant 31$. Kasutada võimalikult vähe sulge.

30. Arvutada interaktiivses keskkonnas list, mille elemendid on omakorda listid, millest vähemalt üks on tühi ja üks ei ole. Kasutada võimalikult vähe sulge.

2.3. Teised infikskonstruktorid. Ehkki tüüpiliselt kohtab konstruktoreid, mille väärtus on funktsioon, andmestruktuuride moodustamisel, võib Haskell pakkuda ka üllatusi.

Argumente võtvaid konstruktoreid võib avastada isegi arvude puhul. Nimelt standardteegi moodul `Data.Complex` annab kompleksarvude esituse kujul `a :+ b`, kus `a` väljendab reaalosa ja `b` imaginaarühiku `i` kordajat imaginaarosas (näiteks arv `i` esitub kujul `0 :+ 1`) ning need mõlemad peavad olema sama ujukomaarvulist tüüpi. Infiksoperaator `:+` on konstruktor. See muidugi tähendab, et Haskellis kujutab kompleksarv endast paarilaadset andmestruktuuri.

Ülesandeid

31. Arvutada interaktiivses keskkonnas i^2.

2.4. Konstruktorite rakendamine. Konstruktorid on tüüpiliselt laisad, st nende rakendamisel jäävad argumendid puutumata. Kui nad olid väärtustamata, siis nad on ka pärast konstruktori rakendamist väärtustamata. Konstruktori rakendamine vaid ehitab argumentide ümber struktuuri, argumentide väärtustamiseks peab olema muu põhjus.

Näiteks listikonstruktori puhul tagab see, et uue elemendi lisamiseks kulub ühepalju aega sõltumata saba pikkusest või elementide suurusest. Ka paarikonstruktor on laisk.

Mõneti erandlikult on konstruktor `:+` agar, mis tähendab, et avaldise `a :+ b` väärtustamisel väärtustatakse kõigepealt `a` ja `b`. Sellise korralduse põhjuseks on asjaolu, et enamasti pole mõtet hoida mälus osaliselt väärtustamata kompleksarve.

3. Tüübitaseme infiksoperaatorid. Infiksoperaatoreid kohtame ka tüübitasemel.

Näiteks paaritüüpi, mille komponenditüübid on A ja B ja mida teoorias kirjutatakse $A \times B$, tähistatakse Haskellis tüübitaseme infiksoperaatori ehk kahe argumendiga tüübikonstruktori abil. Mõneti ebajärjekindlalt on selleks infiksoperaatoriks sulud ja koma sarnaselt paarikonstruktoriga. Näiteks avaldise `('a' , False)` tüüp on `(Char , Bool)`. Paaritüübi segiminekut tüübipaariga pole karta, kuna Haskellis pole kunagi vaja tüübipaare kirjutada.

Analoogselt märgitakse kolmikutüüpi tüüpide kolmiku vormis ja ühiktüüpi tühjade sulgudega kujul `()` nagu tema ainsat väärtustki.

Ka võtmesõna `::` võib tinglikult käsitada infiksoperaatorina, mille vasak argument peab olema andmeavaldis, parem aga tüübiavaldis. Tema prioriteet on madalam ükskõik millise reeglipärase infiksoperaatori omast ning ta on mitteassotsiatiivne.

Ülesandeid

32. Küsida interaktiivses keskkonnas avaldise `(2 , 3)` tüüp ja saada vastusest aru.

33. Annoteerida avaldise `(2 , 3)` tüüp korrektselt kahel viisil: ühel, kus paari komponenditüübid on võrdsed, ja teisel, kus nad on erinevad.

34. Demonstreerida interaktiivses keskkonnas, et võtmesõna `::` on infiksoperaatorina mitteassotsiatiivne ja tema prioriteet on madalam mõne tuntud reeglipärase infiksoperaatori prioriteedist.

35. Kas võtmesõnal `::` oleks mõttekam olla vasak- või paremassotsiatiivne?

4. Infiksoperaatorite tüübid. Olulise täpsustusena tuleb märkida, et kõigi infiksoperaatorite väärtuseks loetakse karritatud funktsioonid. Sellele vastavalt näitab infiksoperaatorite tüüpe ka interaktiivne keskkond. (Kuna paljas infiksoperaator ei moodusta iseseisvat avaldist, tuleb tema tüüpi küsida käsuga `:i`, mitte `:t`.)

Näiteks && väärtuseks olev funktsioon võtab argumendiks mitte tõeväärtuste paari, vaid kaks tõeväärtust ükshaaval. Esimesel neist annab see funktsioon tulemuseks funktsiooni, mis omakorda võtab argumendiks teise tõeväärtuse ja annab siis tulemuseks nende kahe tõeväärtuse loogilise korrutise ehk konjunktsiooni.

Sellest tulenevalt on operaatori && tüüp `Bool -> (Bool -> Bool)`. Siin `->` on funktsioonitüübikonstruktor ja, nagu näha, infiksoperaator. Tema vasak argu-

ment märgib argumenditüüpi ja parem väärtusetüüpi. Seejuures on `->` paremassotsiatiivne, seega võinuks kirjutada ka `Bool -> Bool -> Bool`.

Operaatori `+` tüüp on aga `(Num a) => a -> a -> a`, mis ütleb, et `+` väärtuseks on funktsioon, mis võtab argumendi mingist arvutüübist ja annab tulemuseks funktsiooni, mis võtab argumendi samast arvutüübist (sest tüübimuutuja `a` on sama) ja annab tulemuse jällegi samast arvutüübist.

See, et operaatori rakendamisel tekivad vahetulemustena mingid funktsioonid, ei tähenda, et mälus tekitataks mingit uut koodi, vaid see on lihtsalt sobiv matemaatiline tõlgendus. Arvutusprotsess hakkab nii või teisiti reaalselt peale alles siis, kui kõik selleks vajalikud argumendid on käes.

Karritatud variandid on eelistatud argumentide võtmisele komplektina sellepärast, et viimane tähendaks Haskellis argumentide paigutamist järjenditesse, kuid andmestruktuuride (isegi paaride) moodustamiseks ja neist komponentide kättesaamiseks kulub mõnevõrra lisaressurssi.

Ülesandeid

36. Küsida interaktiivses keskkonnas infiksoperaatorite `*`, `/`, `^`, `^^`, `**`, `||`, `==`, `<` tüübid ja saada neist aru.

2.2.3 Prefiksoperaatorid

Vastanduvalt infiksoperaatoritele nimetame funktsiooniväärtusega muutujat või konstruktorit **prefiksoperaatoriks**, kui ta kirjutatakse oma argumendi ette.

Iseseisvalt käivitatava programmi tegemisel juba kasutasime üht prefiksoperaatorit `print`, mille väärtus on funktsioon, mis argumendil x annab väärtuseks protseduuri, mis kirjutab standardväljundisse x sõnekujul.

Moodulis Prelude on defineeritud väga palju väga erinevaid prefiksoperaatoreid. Näiteks `log` — väärtuseks naturaallogaritm, `exp` — eksponentfunktsioon alusel e, edasi `sin` — siinus, `cos` — koosinus, `tan` — tangens, `asin` — arkussiinus jne, `abs` — absoluutväärtus, `not` — loogiline eitus jpm.

Moodulis Data . Char on defineeritud näiteks operaator `ord` — teisendus sümbolist koodiks — ja `chr` — tema pöördfunktsioon ehk teisendus koodist sümboliks. Samast moodulist tulevate operaatorite `isUpper`, `isLower`, `isSpace` väärtuseks on predikaadid, mis kontrollivad, kas argument on vastavalt suurtäht, väiketäht või tühisümbol. Operaatori `toUpper` väärtus on aga funktsioon, mis võtab

argumendiks sümboli ja kui see on väiketäht, siis annab väärtuseks vastava suurtähe, muidu aga argumendi enda. Samast moodulist saab ka analoogse operaatori `toLower` ja veel palju muid.

1. Prefiksne rakendamine.

1.1. Süntaks. Prefikssel rakendamisel kasutatakse üldiselt eraldajana tühikut. Tühiku asemel on lubatud ka muu mittetühi tühisümbolite jada, mis ei tekita treppimisvigu (tühisümboliteks on peale tühiku näiteks tabulaator ja reavahetus). Kui kas operaator või argument on sulgudes, võib neid eraldav tühisümbolite jada isegi tühi olla.

Programmis (1) kasutasime operaatori `print` ja tema argumendi vahel tühikut.

Sarnaselt näiteks kirjutis `log 5` rakendab operaatorit `log` arvkonstandile `5`. Kuid sama avaldise võiksime saada ka, kirjutades `log(5)` või `(log)5` või `(log)(5)`.

1.2. Prefiksne rakendamine kui infiksoperaator. Prefiksse rakendamise kasutamisel tuleb arvestada, et see seob tugevamini igast infiksoperaatorist ning ahelrakendamist loetakse vasakult paremale. Piltlikult öeldes on prefiksse rakendamise osalisi eraldav tühik (või muu tühisümbolite jada) ise üks infiksoperaator, mille prioriteet on 10 (kõigist kõrgem) ja mis on vasakassotsiatiivne.

Näiteks kui on soov leida siinust arvu $2 + 3$ logaritmist, siis ei saa lihtsamalt kui kirjutada `sin (log (2 + 3))`. Sisemised sulud on vajalikud, sest muidu rakenduks `log` vaid arvkonstandile `2`, välimised aga on vajalikud, sest muidu rakenduks `sin` vaid operaatorile `log`.

Ülesandeid

37. Küsida interaktiivses keskkonnas prefiksoperaatorite `not`, `ord`, `isUpper`, `toUpper` tüübid ja saada neist aru.

38. Kontrollida ujukomaarvuliste arvutuste täpsust, arvutades arvu π mõne arkusfunktsiooni kaudu ja võrreldes muutuja `pi` väärtusega.

39. Kompleksmuutuja funktsioonide teoorias $i^i = e^{-\frac{\pi}{2}}$. Arvutada see arv interaktiivses keskkonnas kahel viisil: kompleksarve kasutades ja kompleksarve kasutamata.

40. Kirjutada avaldis, kus kolm trigonomeetrilist operaatorit oleksid järjest rakendatud (st iga järgmine rakenduks jooksva rakendamise tulemusele).

41. Kontrollida ühe avaldise väärtustamisega, kas sümbol, mis asub kooditabelis A-st 30 sümboli võrra tagapool, on väiketäht.

42. Katsetades interaktiivses keskkonnas paljude erinevate argumentidega, uurida mooduli `Data.Char` mõne ülal mitte kirjeldatud muutuja väärtust.

43. Mis on avaldise `log 0` väärtus?

2. Tähtsamate andmestruktuuridega seonduvad operaatorid.

2.1. Paaridega seonduvad operaatorid. Moodulis Prelude on defineeritud operaatorid `fst` ja `snd`, mille väärtuseks on funktsioonid, mis võtavad argumendiks suvalise paari ja annavad tulemuseks vastavalt selle paari esimese ja teise komponendi.

Näiteks `fst (2 , -3)` väärtus on 2, aga `snd (2 , -3)` väärtus on -3.

Kuna paari komponendid võivad olla suvalist tüüpi, on operaatorid `fst` ja `snd` polümorfsed.

Kummagi operaatori rakendamisel seisneb arvutus vaid paari järgi komponendi (asukoha) leidmises, komponendi sisusse ei tungita ja ei puututa ka paari seda komponenti, mida välja andma ei pea. Seega on `fst` ja `snd` rakendamisel kuluv aeg ühesugune sõltumata paarist.

Näite paare konstrueerivast funktsioonist annab samuti moodulis Prelude defineeritud operaator `properFraction`. Tema väärtuseks on funktsioon, mis võtab argumendiks murdarvutüüpi arvu ja annab väärtuseks paari tema täis- ja murdosaga, kus positiivse argumendi korral tuleb mittenegatiivne murdosa ja negatiivse argumendi korral mittepositiivne murdosa.

Ülesandeid

44. Küsida interaktiivses keskkonnas prefiksoperaatorite `fst`, `snd`, `log`, `abs`, `properFraction` tüübid ja saada neist aru.

45. Kirjutada interaktiivses keskkonnas avaldis, kus `fst` ja `snd` esinevad järjest rakendatuna ja mille väärtustamine lõpetab normaalselt.

46. Sisestada interaktiivse keskkonna käsurealt avaldis, mille väärtus väljendab arvu e^{10} murdosa. Mis juhtub, kui astendaja on suurem, nt 100?

2.2. Listidega seonduvad operaatorid. Listide valdkonnas mainime kõigepealt operaatoreid `head` ja `tail`, mille väärtuseks on funktsioonid, mis võtavad argumendiks listi ja kui see on mittetühi, siis annavad tulemuseks vastavalt selle listi pea ja saba.

Näiteks avaldise `head (1 : 0 : [])` väärtus on arv 1, seevastu avaldise `tail (1 : 0 : [])` väärtus on list, mille ainus element on 0.

Operaatori `null` väärtus on predikaat, mis kontrollib, kas argumentlist on tühi.

Operaatorite `head` ja `tail` rakendamiseks kuluv aeg on pikemate ja lühemate listide korral sama, sest nende töö on analoogiline paarioperaatoritega `fst` ja `snd` (leitakse andmestruktuuri väljad — listi pea ja saba, kuid neid uurima ei hakata). Ka `null` rakendamine toimub fikseeritud ajapiirides.

Ainuüksi moodul Prelude annab programmeerijale väga palju listidega seonduvaid operaatoreid. Kõige lihtsamad neist lisaks seniõpituile on `length`, `sum` ja `product`. Nende väärtuseks olevad funktsioonid võtavad argumendiks listi ja kui see on lõplik, siis annavad tulemuseks vastavalt tema pikkuse ehk elementide arvu, elementide summa ja elementide korrutise. Esimene neist funktsioonidest on tüübikorrektselt rakendatav igasugustele listidele, ülejäänud kaks ainult arvulist tüüpi elementidega listidele.

Näiteks avaldise `length (1 : 0 : [])` väärtus on 2, sest `1 : 0 : []` väljendab 2-elemendilist listi, avaldise `product (1 : 2 : 3 : [])` väärtus on aga 6.

Operaatorite `minimum` ja `maximum` väärtuseks on analoogilised funktsioonid, mis lõplikule mittetühjale listile rakendatult annavad välja vastavalt tema minimaalse ja maksimaalse elemendi. Need on tüübikorrektselt rakendatavad kõigi järjestusega elemenditüüpide korral.

Veel on kasulik tunda operaatorit `reverse`, mille väärtuseks on funktsioon, mis lõplikule listile rakendades annab tulemuseks listi samade elementidega vastupidises järjekorras. See funktsioon rakendub korrektselt mistahes tüüpi elementidega listidele.

Operaatorite `length`, `sum`, `product`, `minimum`, `maximum`, `reverse` rakendamine toimub argumentlisti pikkuse suhtes lineaarse ajaga.

Kui `head` ja `tail` väärtuseks olevad funktsioonid jagavad listi osadeks tema esimese elemendi järelt, siis operaatorite `last` ja `init` väärtuseks on analoogsed

funktsioonid, mis jagavad mittetühja argumentlisti osadeks tema viimase elemendi eest, andes välja vastavalt tema viimase elemendi ja listi tema elementidest kuni eelviimaseni. Kuid erinevalt operaatoritest `head` ja `tail` kulub masinal `last` ja `init` rakendamiseks argumentlisti pikkusega võrdeline aeg. Erinevus tuleb sellest, et viimase elemendi ülesleidmiseks on vaja list algusest lõpuni läbi käia.

Hulk listifunktsioone, mida moodulist Prelude ei saa, on kättesaadavad standardteegi mooduli Data.List kaudu. Seal on näiteks operaatori `tails` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks listi, mille elementideks on argumentlist, tema saba, saba saba jne, kuni tühja listini, st argumentlisti kõik alamlistid. Analoogiline funktsioon on operaatori `inits` väärtuseks: tema võtab samuti argumendiks listi ja annab tulemuseks listi, mille elementideks on pikkuse kasvamise järjestuses tema kõik algusjupid alates lühemast, kaasa arvatud tühi list. Mõlemad funktsioonid töötavad ka tühjal listil.

Kui `tails` rakendamiseks kuluv aeg on võrdeline argumentlisti pikkusega, siis `inits` rakendamine nõuab aega koguni võrdeliselt argumentlisti pikkuse ruuduga. Vahe tuleb sisse sellest, et esimesel juhul on konstrueeritava listi kõik elemendid argumentlisti struktuuris olemas, nende aadressid on vaid vaja kokku koguda, kuid argumentlisti algusjupid, mida vajab `inits`, on vaja ka valmis ehitada.

Ülesandeid

47. Testida operaatoreid `head`, `tail`, `null`, `length`, `sum` interaktiivses keskkonnas, koostades igaühe jaoks neist avaldise, milles ta on rakendatud mingile argumendile, mille väärtus on vähemalt 3-elemendiline list, nii et avaldise väärtustamine lõpeb normaalselt.

48. Kirjutada interaktiivses keskkonnas üks avaldis, milles esinevad mingis järjekorras järjest rakendatuna muutujad `head`, `sum` ja `reverse` ja mille väärtustamine lõpeb normaalselt.

49. Arvutada interaktiivses keskkonnas niisugune list, mille elementideks on listi $1:3:6:10:15:21:28:[]$ kõik mittetühjad algusjupid pikkuse kasvamise järjestuses.

50. Arvutada interaktiivses keskkonnas niisugune list, mille elementideks on listi $1:3:6:10:15:21:28:[]$ kõik mittetühjad lõpujupid pikkuse kasvamise järjestuses.

2.3. Lõpmatute andmestruktuurite koostamine ja testimine. Haskellis võib avaldise väärtus olla ka lõpmatu andmestruktuur.

Operaatori `repeat` väärtuseks on funktsioon, mis võtab suvalise argumendi x ja annab väärtuseks lõpmatu listi, mille iga element on x.

Kui vaja huvitavamat listi, võib kasutada operaatorit `cycle`. Tema väärtuseks on funktsioon, mis suvalisel lõplikul mittetühjal listil l annab välja lõpmatu listi, kus l elemendid korduvad tsükliliselt. Näiteks `cycle (1 : 2 : 1 : [])` väärtus on lõpmatu list elementidega $1, 2, 1, 1, 2, 1, 1, 2, 1, \ldots\ldots$.

Lõpmatu struktuuri täielik väärtustamine on muidugi võimatu. Selline protsess ei peatu enne, kui miski väline teda katkestab. Interaktiivses keskkonnas saab arvutusprotsessi katkestada, vajutades <*Ctrl*>+`c`.

Kummatigi saab selliseid avaldisi konteksti asetatuna praktikas edukalt kasutada, sest konstruktori laiskuse tõttu tehakse väärtustusprotsessis andmestruktuuridest valmis parajasti niipalju kui hetkel vaja. Lõpmatu andmestruktuur avaldise väärtusena tähendab, et lõpmatu väärtustamise käigus on lõpliku ressursiga põhimõtteliselt võimalik kätte saada listi kuitahes kauge fragment.

Intelligentne meetod lõpmatu andmestruktuuri testimiseks on küsida tema erinevaid lõplikke osi. Listi üksikute elementide kättesaamiseks sobib infiksoperaator `!!`. Kui `l` väärtuseks on list l ja `a` väärtuseks naturaalarv a, mis on väiksem l pikkusest, siis `l !! a` väärtus on l element järjekorranumbriga a. Seejuures nummerdatakse listi elemente alates 0-st. Kui `a` väärtus pole nõutud piirides, siis tekib täitmisaegne viga.

Näiteks `(1 : 2 : 3 : []) !! 1` väärtus on 2. Avaldise `repeat 0 !! a` väärtus on alati 0, kui vaid `a` väärtuseks on naturaalarv.

Tuleb arvestada, et listi elemendi leidmine operaatoriga `!!` toimub järjekorranumbri suhtes lineaarses ajas.

Lõpmatud listid on mugavad, sest võimaldavad tihti vältida ülesande seisukohalt kunstliku piiri sissetoomist. Seetõttu on Haskellis programmeerimisel lõpmatute listide kasutamine standardne. Näiteks võib naturaalarvuliste argumentidega funktsiooni arvutamiseks kirjeldada lõpmatu listi selle funktsiooni väärtustega kõigil argumentidel ja funktsioonil lihtsalt lasta sealt lugeda.

Haskellis programmeerides puutume tihti kokku ka lõpmatute listidega teiste andmestruktuuride koosseisus. Testides niisugust andmestruktuuri lohakalt, on oht osa struktuurist ära kaotada. Kuid see on välditav.

Näiteks avaldise `(repeat 1 , repeat 2)` väärtus on paar, mille kumbki komponent on lõpmatu list. Kui laseksime seda paari lihtsalt käsurealt väärtustada, näeksime ainult ühtesid, arvutus ei jõuaks kunagi paari teise komponendini. Ometi

on ka kahed olemas, selles veendumiseks piisab lihtsalt võtta struktuurist teine komponent: `snd (repeat 1 , repeat 2)` annab tulemuseks kahtedest koosneva listi.

Ülesandeid

51. Teha kindlaks operaatori `!!` prioriteet ja assotsiatiivsus.

52. Väärtustada interaktiivse keskkonna käsurealt avaldis, mille väärtus on lõpmatu list. Katkestada arvutus ja seejärel täiendada avaldist nii, et saadud avaldise väärtustamine lõpeb kiiresti.

53. Kirjutada interaktiivses keskkonnas avaldis, mille väärtus on lõpmatu list, mille esimesed elemendid on 1 ja 2 ning kõik ülejäänud elemendid on 0-d. Testida seda listi tema üksikuid elemente küsides.

54. Kirjutada interaktiivses keskkonnas avaldis, milles esineb alamavaldisena `(repeat 1 , repeat 2)` ja mille väärtustamine lõpeb normaalselt.

55. Katseliselt veenduda, et `!!` abil listi elemendi leidmiseks kulub järjekorranumbriga võrdeline aeg.

56. Milline on muutujate `length`, `sum`, `product`, `minimum`, `maximum`, `reverse`, `last`, `init`, `tails`, `cycle` väärtuseks olevate funktsioonide rakendamise tulemus (a) tühjal listil, (b) lõpmatul listil, (c) osalisel listil?

3. Karritatud väärtusega prefiksoperaatorid. Ka prefiksoperaatori väärtus võib olla karritatud funktsioon, st süüa pärast üht argumenti veel argumente, kuni tulemus pole enam funktsioon. Valdav enamik mitmeparameetrilisi funktsioone standardteegis ongi realiseeritud karritatult.

Operaatori `const` väärtus on üks lihtsaimaid võimalikke karritatud funktsioone. See annab oma argumendil x välja funktsiooni, mis omakorda suvalisel oma argumendil annab väärtuseks x. Niisiis kahest argumendist annab ta välja esimese ehk ta on saadud paarist esimese komponendi välja andva funktsiooni karritamisel.

3.1. Karritatud funktsioonide kasutamine. Muutujast-konstruktorist keerulisemate avaldiste rakendamine toimub ainult prefiksselt. Seetõttu tuleb karritatud väärtusega prefiksoperaatori rakendamisel kõik argumendid kirjutada järjest operaatori järele.

Kuna prefiksne rakendamine on vasakassotsiatiivne, st `a b c = (a b) c`, siis sulge pole vaja mujal kui võibolla argumentide ümber nende eristamiseks. See on veel üks omadus, mille tõttu Haskellis programmeerimisel eelistatakse karritatud funktsioone.

Näiteks operaatori `const` rakendamiseks argumendile `5` kirjutame nagu ikka `const 5`. Selle avaldise väärtus on funktsioon, seega saame veel korra rakendada — näiteks argumendile `0.2`, millega saame avaldise `(const 5) 0.2`. Vastavalt `const` ülalkirjeldatud definitsioonile on viimase avaldise väärtus 5. Rakendamise vasakassotsiatiivsuse tõttu võib sulud ära jätta ja kirjutada `const 5 0.2`.

Vaadeldud karritatud funktsiooni juures on huvitav märkida, et tema väärtus normaalsel argumendil on alati laisk funktsioon. Et seda näha, võib interaktiivses keskkonnas anda väärtustada näiteks avaldise `const 0 undefined`. Vastuseks tuleb `0`. Argumendi `undefined` antav potentsiaalne veateade jääb olemata, järelikult seda argumenti ei püütagi väärtustada.

3.2. Suurustega seonduvad karritatud funktsioonid. Suuruste võrdlemisega seonduvad operaatorid `min` ja `max`. Nende väärtuseks on karritatud funktsioonid, mis võtavad kaks argumenti samast järjestusega tüübist ja annavad välja vastavalt neist kahest minimaalse ja maksimaalse.

Seega näiteks `min 7 2` väärtus on 2, aga `max 7 2` väärtus 7. Üldiselt on avaldis `min` e_1 e_2 on alati väärtuselt võrdne avaldisega `minimum (`e_1 `:` e_2 `: [])`.

Operaatorite `div` ja `mod` väärtuseks on karritatud funktsioonid, mis võtavad argumentidena kaks arvu ja annavad väärtuseks vastavalt täisarvulise jagatise ja jäägi esimese arvu jagamisel teisega.

Näiteks avaldiste `div 10 7` ja `mod 10 7` väärtus on vastavalt 1 ja 3, sest 10 jagamisel 7-ga on jagatis 1 ja tekib jääk 3.

Kui on vaja nii jagatist kui ka jääki, siis võib kasutada operaatorit `divMod`, mille väärtus on funktsioon, mis annab välja oma argumentide jagatise ja jäägi ühes paaris koos. Avaldise `divMod 10 7` väärtus on paar $(1, 3)$.

3.3. Listidega seonduvad karritatud funktsioonid. Palju kasutatakse operaatoreid `take` ja `drop`. Kummagi väärtuseks on karritatud funktsioon, mis võtab sisse lühikese täisarvu n ja listi l. Tulemuseks on `take` puhul list, mis koosneb listi l esimesest n elemendist järjekorda muutmata, kui l-s on vähemalt n elementi, ja tervest listist l, kui l-s on vähem kui n elementi, `drop` puhul aga nende elementide list, mis `take` puhul üle jäävad.

Näiteks `take 2 (5 : 6 : 7 : [])` väärtus on lõplik list elementidega 5, 6 ja `drop 2 (5 : 6 : 7 : [])` väärtus on lõplik list elemendiga 7.

Analoogselt jäägiga jagamisega on ka siin olemas võimalus arvutada mõlemad osad korraga, operaatoriga `splitAt`. Sestap `splitAt 2 (5 : 6 : 7 : [])` väärtus on list elementidega 5, 6 paaris listiga, mille ainus element on 7.

Vaadeldud operaatorite rakendamiseks kuluv aeg on võrdeline nende esimese argumendiga. Kui listi elementide arv on suurem, ei puutu ülejäänud elemendid asjasse. Seetõttu sobib operaator `take` hästi lõpmatute listide algusjuppide küsimiseks.

Näiteks avaldise `take 3 (repeat 2)` väärtus on lõplik list elementidega 2, 2, 2 ja väärtustamisprotsess käib välgukiirusel.

Kindla pikkusega listide konstrueerimiseks võib kasutada operaatorit `replicate`, mille väärtuseks on karritatud funktsioon, mis võtab argumentideks lühikese täisarvu a ja suvalise andme x ning annab välja listi pikkusega a, mille iga element on x. Selle operaatori rakendamine kulutab tema täisarvargumendiga võrdelise aja.

Mõnikord osutub kasulikuks muutuja `zip`, mille väärtuseks on karritatud funktsioon, mis võtab argumendiks kaks listi ja annab välja paaride listi, kus paarideks on kokku võetud argumentlistide vastavad elemendid. Argumentlistidest pikema ülejääv sabaosa rolli ei mängi.

Näiteks avaldise `zip (1 : 2 : []) (3 : 4 : 5 : 6 : [])` väärtus on list, mille elemendid on paarid $(1\,,3)$ ja $(2\,,4)$.

Ülesandeid

57. Küsida interaktiivses keskkonnas muutujate `const`, `min`, `max`, `div`, `mod`, `divMod` tüübid ja saada neist aru.

58. Väärtustada interaktiivses keskkonnas avaldis, mille väärtuseks on 500-elemendiline tõeväärtuste list.

59. Väärtustada interaktiivses keskkonnas avaldis, mille väärtus on 1001-elemendiline list, mille esimesed 1000 elementi võrduvad 0-ga ja viimane on 1.

60. Väärtustada interaktiivses keskkonnas avaldis, mille väärtus on list, mille elementideks on listid 19, 18, jne, kuni 10 elemendiga 1.

61. Kirjutada võimalikult lühike avaldis, mille väärtus on list, mille elementideks on listi $1:3:6:10:15:21:28:[]$ vähemalt 3-elemendilised algusjupid pikkuse järgi kasvavas järjestuses.

4. Prefikskonstruktorid. Kõik senivaadeldud prefiksoperaatorid on muutujad, kuid muidugi on infiksselt rakendatavate konstruktorite kõrval olemas ka prefiksselt rakendatavad konstruktorid.

Kasutatavaim prefiksselt rakendatav konstruktor on `Just`. Tema väärtus on funktsioon, mis võtab argumendiks suvalise andme x mingist tüübist A ja annab väärtusena välja Just x nurjumisega tüübist Maybe A. Nurjumisega tüüpidega seondub veel konstruktor `Nothing`, mis argumente ei võta (väärtus pole funktsioon).

Mitmesuguseid kasulikke nurjumisega tüüpidega seonduvaid operaatoreid saab standardteegi moodulist Data . Maybe.

Ülal vaatlesime muutujat `head`, väärtuseks listi järgi tema pea leidmise funktsioon. Kui argumendiks juhtub tulema tühi list, siis pead ei ole, pole võimalik midagi vastuseks anda — `head []` väärtustamine lõpetab täitmisaegse veaga. Moodulist Data . Maybe leiab aga muutuja `listToMaybe`, mille väärtus on funktsioon, mis annab argumentlisti elemendi asemel välja nurjumisega tüüpi elemendi: kui argumentlist on mittetühi ja tema pea on x, siis Just x, tühja argumentlisti puhul Nothing. See funktsioon on sisult sarnane, kuid ei tekita kunagi täitmisaegset viga.

Operaatori `fromMaybe` väärtus on karritatud funktsioon, mis võtab argumentideks mingi suvalise andme ja nurjumisega tüüpi andme. Kui viimane on kujul Just x, antakse väärtuseks x, vastasel korral esimene argument. Funktsiooni ülesanne on kaotada teise argumendi Just, kuid kui see on Nothing, läheb käiku varuna antud esimene argument. Juhul, kui nurjumist ei kardeta, võib Just eemaldada ka operaatori `fromJust` abil. Selle muutuja väärtuseks on funktsioon, mis argumendil Just x annab väärtuseks x, argumendil Nothing aga normaalset väärtust ei ole (`fromJust Nothing` väärtustamine lõpeb veateatega).

5. Alternatiivvõimalus prefiksseks rakendamiseks. Oleme funktsiooni rakendamist märkivat tühikut (või tühisümbolite jada) tinglikult käsitanud infiksoperaatorina. Prefiksse rakendamise jaoks on olemas ka päris infiksoperaator `$`.

Näiteks avaldise `log 5` asemel võib samaväärselt kirjutada `log $ 5`.

Operaatori `$` prioriteet ja assotsiatiivsus on tavalise rakendamisega võrreldes vastupidised: ta on paremassotsiatiivne prioriteediga 0. Sellisena on `$` kasulik järjestrakendamisel.

Näiteks avaldise

```
take 20 (tails (replicate 100 (9 ^ 9)))
```

asemel võib kasutada samaväärset sulgudeta avaldist

```
take 20 $ tails $ replicate 100 $ 9 ^ 9.
```

Pangem tähele, et `$` väärtus on kõrgemat järku funktsioon, kuna võtab argumendiks funktsiooni.

6. Tüübitaseme prefiksoperaatorid. Loomulik mõte on, et tüübifunktsioonid, mille liik on $* \to *$, võiksid olla realiseeritud prefiksoperaatoritena. See oletus peab valdavalt ka paika.

6.1. Tähtsamad tüübikonstruktorid. Näiteks tüübifunktsiooni List märgib Haskellis tüübikonstruktor `[]`, tüübifunktsioone Maybe ja IO aga vastavalt tüübikonstruktorid `Maybe` ja `IO`.

Avaldise `1 : 0 : []` tüübiks võib kirjutada näiteks `[] Int` või `[] Double` või ka üldiselt `(Num a) => [] a`, avaldise `Just pi` tüübiks võib panna `Maybe Float` või `Maybe Double` või `(Floating a) => Maybe a`.

Lisaks sellele, et konstruktor `[]` on erandliku kujuga — ta ei vasta leksikanõuetele —, ei käsitle interaktiivsed keskkonnad kasutajale vastuseid koostades teda üldse prefiksoperaatorina, vaid täiesti erilaadsena, asetades argumendi kantsulgude sisse, mitte järele.

Ülesandeid

62. Küsida interaktiivses keskkonnas konstruktorite `[]` ja `:` ning muutujate `head`, `tail`, `null`, `length`, `sum`, `product`, `minimum`, `maximum`, `reverse`, `tails`, `inits`, `repeat`, `cycle`, `take`, `drop`, `splitAt`, `replicate`, `zip` tüübid ja saada neist aru.

63. Küsida interaktiivses keskkonnas konstruktorite `Just`, `Nothing` ning muutujate `listToMaybe`, `fromMaybe`, `fromJust`, `print` tüübid ja saada neist aru.

64. Kirjutada korrektne tüübiannotatsiooniga avaldis, mille tüüp on listitüüp, mille elementide tüüp on mingi nurjumisega tüüp.

6.2. Kitsendatud argumendipiirkonnaga operaatorid. Seni vaadeldud tüübifunktsioonid võisid võtta suvalisi tüüpe argumendiks. Kuid on võimalikud ka teistsugused.

Keerulisemate arvutüüpidega seonduvad tüübikonstruktorid saavad argumendiks võtta vaid teatavaid lihtsamaid arvutüüpe.

Kompleksarvutüüp sõltub sellest, millist ujukomaarvutüüpi on väljad. Moodulis Data.Complex defineeritud tüübikonstruktori `Complex` väärtuseks on funkt-

sioon, mis võtab argumendiks väljatüübi A, mis peab olema klassist Floating, ja annab tulemuseks kompleksarvutüübi, mille väljatüüp on A.

Näiteks avaldise `0.8 :+ 0.6` tüübiks võib kirjutada `Complex Float`, samuti `Complex Double` või `(Floating a) => Complex a`.

Analoogselt on olemas ka mitu ratsionaalarvutüüpi. Ratsionaalarvu esitatakse andmestruktuurina lugejast ja nimetajast, mis peavad olema ühesugust tüüpi täisarvud. Moodul Data . Ratio annab tüübikonstruktori `Ratio` väärtuseks funktsiooni, mis võtab argumendiks täisarvutüübi A ja annab tulemuseks ratsionaalarvutüübi, kus murdude lugeja ja ja nimetaja on tüüpi A. Niisiis on vaikimisi võimalikud ratsionaalarvutüübid `Ratio Int` ja `Ratio Integer`.

6.3. Tüübisünonüümid. Haskell lubab tüübitasemel ka globaalseid muutujaid, st ühest nimest koosnevaid tüübiavaldisi, mis ei esinda iseennast. Selliste nimed peavad algama suurtähega. Tavaliselt nimetatakse sellist tüübimuutujat tüübisünonüümiks.

Väga kasutatav tüübisünonüüm on `String`, mis on samaväärne tüübiavaldisega `[] Char`, st sõned on sümbolite listid.

Ka varem vaadeldud `Rational` on tüübisünonüüm, globaalne tüübimuutuja, mida võib samaväärselt ümber kirjutada avaldisega `Ratio Integer`.

2.2.4 Nime prefikskuju ja infikskuju

1. Nimekujud andmetasemel. Infiksoperaatorite tüüpide seletamisel sai möödaminnes märgitud, et paljas infiksoperaator ei moodusta omaette avaldist. See võib tunduda funktsionaalse paradigma rikkumisena, sest funktsionaalse keele süntaks ei tohiks teha funktsioonide ja muude andmete vahel vahet. Tegelikult rikkumist ei ole.

Asi on selles, et kõigil andmemuutujatel ja -konstruktoritel on põhimõtteliselt olemas kaks nimekuju: üks on infiksseks kasutamiseks, teine muudes olukordades kasutamiseks. Ütleme nende kujude kohta vastavalt infikskuju ja prefikskuju. Mistahes muutuja või konstruktor moodustab omaette avaldise küll, aga ta tuleb selleks esitada prefikskujul, sest omaette avaldisena pole kasutus infiksne.

1.1. Kuju määramine ja teisendamine. Nimekirja (3) sümbolitest koosnevad nimed on kõik infikskujul ja et neid kasutada mitteinfiksselt, tuleb nad asetada sulgudesse.

Näiteks `*` prefikskuju on `(*)`. Paljas `*` ei ole avaldis, kuid `(*)` on.

Tähtedest, numbritest ja alakriipsust koosnevad nimed on prefikskujul ja et neid kasutada infiksselt, tuleb nad panna tagurpidiülakomade vahele.

Näiteks `mod` infikskuju on `` `mod` ``.

Avaldis `replicate 3 0.5 ++ replicate 4 0.8` on sama mis avaldis `(++) (3 `replicate` 0.5) (4 `replicate` 0.8)`. Siin on infiksoperaator `++` kasutatud prefiksselt ja prefiksoperaator `replicate` infiksselt.

Niisiis iga infiksoperaatori saab teha prefiksoperaatoriks ja, vähemalt süntaktiliselt, saab iga prefiksoperaatori teha infiksoperaatoriks.

Muidugi saab muutuja või konstruktori infiksne kasutus mõttekas olla vaid juhul, kui tal leidub kaks argumenti, mille vahele ta panna, st kui tema väärtuseks on karritatud funktsioon. Sellisel juhul vastab prefiksse rakendamise esimene argument infiksse rakendamise vasakule argumendile ja prefiksse rakendamise teine argument infiksse rakendamise paremale argumendile. Vastasel korral annab infiksne rakendamine tüübivea.

Kõik infikssed rakendamised kirjutab Haskelli süntaksianalüsaator automaatselt prefikssete rakendamiste kaudu ümber.

Ülesandeid

65. Kirjutada avaldises `2 + 3 + 6` infikssed rakendamised prefikssetena ümber.

66. Kirjutada avaldises `2 ^ 3 ^ 6` infikssed rakendamised prefikssetena ümber.

67. Kirjutada avaldises `4 * 0.3 : 4 - 0.5 : []` infikssed rakendamised prefikssetena ümber.

68. Arvutada nii jagatis kui jääk arvu 10000000 jagamisel arvuga 4649, lastes väärtustada vaid ühe võimalikult lühikese avaldise, kus prefiksset rakendamist ei esine.

69. Nagu ülal väidetud, saab iga infiksoperaatori teha prefiksoperaatoriks ja vastupidi. Kirjeldada ammendavalt, kuidas.

1.2. Prioriteet ja assotsiatiivsus. Prefiksoperaatorist tagurpidiülakomade abil tuletatud infiksoperaatorid on, nagu ülejäänudki, vaikimisi prioriteediga 9 vasakassotsiatiivsed, kuid mõne operaatori jaoks on need atribuudid üle defineeritud.

Näiteks muutujate `div` ja `mod` infikskujud on korrutamise-jagamisega sama prioriteediga 7 vasakassotsiatiivsed.

Ülesandeid

70. Määrata muutuja `divMod` infikskuju prioriteet ja assotsiatiivsus.

2. Nimekujud tüübitasemel. Mis puutub tüübimuutujatesse ja -konstruktoritesse, siis infiksoperaatoritel on siingi prefikskuju olemas.

Näiteks tüübiavaldist `Int -> Char` saab kirjutada ka kujul `(->) Int Char`.

Infikskuju moodustamine tähtedest, numbritest ja alakriipsust koosneva nimega tüübioperaatoritest on Haskelli standardis keelatud. GHC laiendustes on aga ka see võimalik.

3. Adresseeritud nimede kujud. Eksporditakse ja imporditakse nimesid koos oma tähendusega, mitte kujusid. Kujuteisendus on puhtlokaalne ettevõtmine. (Ekspordi- ja impordiloendis kasutatakse prefikskuju.)

Adresseerimisel lähtutakse nime algkujust ja kujuliik seejuures ei muutu.

Näiteks kirjutis `Data.Maybe.fromMaybe` on prefikskuju; vastav infikskuju on `‘Data.Maybe.fromMaybe‘`. Samas `Data.Ratio.%` on infikskuju ja vastav prefikskuju on `(Data.Ratio.%)`.

2.2.5 Sektsioonid

1. Sektsioonide liigid ja süntaks. Andmetaseme infiksoperaatoritest saab spetsiaalse süntaktilise konstruktsiooniga moodustada nn sektsioone. Sektsioon kujutab endast teatavat avaldist, mille väärtus on funktsioon. Sektsioonid jagunevad vasak- ja paremsektsioonideks.

1.1. Üldreegel. Vasaksektsioon näeb välja kujul

$$\texttt{(}\mathfrak{a}\ \oplus\texttt{)},$$

kus $\mathfrak{a}$ on avaldis ja $\oplus$ on infiksoperaator. Sellise vasaksektsiooni väärtus on funktsioon, mis suvalisel argumendil b annab väärtuseks $f\,a\,b$, kus f on infiksoperaatori $\oplus$ väärtus ja a avaldise $\mathfrak{a}$ väärtus.

Analoogselt näeb paremsektsioon välja kujul

```
(⊕ 𝔟)
```

kus $\oplus$ on infiksoperaator ja $\mathfrak{b}$ avaldis. Sellise avaldise väärtus on funktsioon, mis igale argumendile a seab vastavusse $f\ a\ b$, kus f on $\oplus$ väärtus ja b on $\mathfrak{b}$ väärtus.

Teisi sõnu, sektsiooni rakendamine argumendile on samaväärne selles sektsioonis esineva infiksoperaatori rakendamisega sellele argumendile ja sektsiooni sees olevale avaldisele, kusjuures sektsiooni sees olev avaldis jääb operaatorist samale poole nagu ta on sektsioonis.

Näiteks `(* 2)` väärtus on funktsioon, mis korrutab oma argumendi 2-ga. Sama väärtusega on ka `(2 *)`, sest korrutamine on kommutatiivne.

Avaldiste `(/ 2)` ja `(2 /)` väärtused on aga erinevad: esimene on funktsioon, mis jagab oma argumendi 2-ga, teine on funktsioon, mis jagab arvu 2 oma argumendiga.

Näiteks `(* 5)` 7 väärtus on 35, `(3 ^)` 2 väärtus 9 ja `(^ 3)` 2 väärtus 8.

1.2. Miinusmärgi erand. Kui vasaksektsioone võib moodustada suvalise infiksoperaatoriga, siis paremsektsiooni moodustamine on lubatud kõigi infiksoperaatoritega peale miinuse. Miinus on keelatud, sest kirjutis kujul `(- 𝔞)` läheks segi vastandarvu leidmist tähistava miinusega.

Olukorras, kus oleks vaja miinuse paremsektsiooni, aitab hädast välja muutuja `subtract`, mille väärtus on karritatud funktsioon, mis arvulisel argumendil x annab väärtuseks funktsiooni, mis oma argumendist lahutab x. Näiteks `subtract 2 3` väärtus on 1. Puuduva sektsiooni `(- 𝔞)` asemel tuleb kasutada avaldist `subtract 𝔞`.

Ülesandeid

71. Kirjutada ülesannetes 65 ja 66 toodud avaldistes infikssed rakendamised vasaksektsioonidega ümber.

72. Kirjutada ülesannetes 65 ja 66 toodud avaldistes infikssed rakendamised paremsektsioonidega ümber.

73. Selgitada võimalikult lihtsate sõnadega, mis on avaldise `(: [])` väärtus.

74. Teha kindlaks, kas avaldiste

    `(True ||)`, `(False &&)`, `(|| True)`, `(&& False)`

    väärtuseks olevad funktsioonid on agarad või laisad.

2. Sektsiooni praktiline kasutus. Infiksset rakendamist sektsiooniga ümber kirjutada pole muidugi mõtet. Sektsioonid on mugavad olukordades, kus infiksoperaatoril on ainult üks argument fikseeritud, teine on lahtine.

Vaatleme operaatorit `map`, mille väärtuseks on karritatud funktsioon, mis võtab argumentideks funktsiooni f ja listi l ning annab tulemuseks listi, mille elemendid on saadud listi l vastavatele elementidele funktsiooni f rakendamisel. Avaldises, kus rakendatakse operaatorit `map`, ei esine ühtki tema argumentfunktsiooni rakendamist, argumentfunktsiooni argument jääb lahtiseks.

Võime seda näha avaldise `map (^ 2) (5 : 6 : 7 : 8 : [])` peal. Tema väärtus on arvude 5, 6, 7, 8 ruutude list, kuid ruutfunktsiooni kodeeriv sektsioon `(^ 2)` pole avaldises rakendatud ühelegi argumendile.

Tähelepanelik lugeja märkab, et `map` väärtus on kõrgemat järku funktsioon.

Kui funktsioonitüüpi avaldisele on antud niipalju argumente, et tulemus pole enam funktsioonitüüpi, siis öeldakse, et ta on täisargumenteeritud. Oleme seega näinud, et Haskellis on korrektsed ja praktilised ka täisargumenteerimata avaldised.

Ülesandeid

75. Küsida interaktiivses keskkonnas muutuja `map` tüüp ja saada sellest aru.

76. Lahendada ülesanne 28, kasutades sektsioone ja muutujat `map`.

2.3 Listide erisüntaksid

Kuna listid on praktilises funktsionaalses programmeerimises põhilised andmestruktuurid, on Haskellis mitu listidega seonduvat erisüntaksit. Peale selles jaotises vaadeldavate on olemas veel listikomprehensioon, mille tema suhtelise keerulisuse pärast jätame hilisemaks.

2.3.1 Interaktiivse keskkonna kasutatavad erisüntaksid

Kolme listidega seonduvat erisüntaksit kasutavad ka interaktiivsed keskkonnad info esitamisel kasutajale.

1. Listitüübi erisüntaks. Listitüüpide esitus kujul, kus tüübikonstruktori `[]` argument on konstruktori sees kantsulgude vahel, kuulub Haskelli süntaksisse, st nii saab ka koodis kirjutada.

Niisiis on näiteks `[Integer]` ja `[Bool]` tüübiavaldised, mis väljendavad tüüpe, kuhu kuuluvad vastavalt täisarvuliste elementidega listid ja tõeväärtusetüüpi elementidega listid.

2. List elementide loendina. Listi esitamine komaga eraldatud elementide loendina kantsulgudes kuulub samuti Haskelli süntaksisse.

Näiteks `[pi]` on avaldis, mis on samaväärne avaldisega `pi : []`, ning samuti `[1, 0, -1]` on avaldisega `1 : 0 : -1 : []` samaväärne avaldis.

Nii saab liste esitada pisut lühemalt kui koolonite kaudu. Kuid selle süntaksi kasutamine pole kaugeltki alati võimalik. Listi esitamine elementide loendina eeldab, et listi struktuur on kodeerimise ajal lõpuni teada, sest taoline esitus näitab ära listi täpse pikkuse. See tingimus on aga praktikas harva täidetud. Kunagi ei ole võimalik elementide loendina esitada lõpmatuid või osalisi liste.

3. Sõnekonstandid. Kuna sõned on listid, on neid võimalik kodeerida kõigi sobivate listisüntaksite abil. Lubatud on ka sõnekonstandid — sümbolite jorud jutumärkide vahel, mida näeme sümbolite listide puhul interaktiivses keskkonnas. Seejuures on kasutatavad kõik sümbolite langjoonkoodid. Apostroofi kodeerimine kujul `\'` pole kohustuslik, küll aga jutumärgi kodeerimine kujul `\"`.

Sõnekonstant ei saa sisaldada muutujaid. Sõnekonstandi kasutamine seega eeldab, et terve sõne koos oma sümbolitega on kodeerimise ajal teada.

Ülesandeid

77. Kirjutada interaktiivse keskkonna käsurealt avaldisi, mille väärtuseks on sõne “Tere!”. Kasutada kolme erinevat süntaktilist konstruktsiooni.

3.1. Veateated. Sõnekonstantide võibolla et levinuim kasutus on veateadete kirjutamisel. Veateadete tekitamiseks on olemas operaator `error`, mille väärtus on funktsioon, mis võtab argumendiks suvalise sõne s, kuid ei anna mingit normaalset väärtust, operaatori rakendamine lõpeb veateatega s.

Andes näiteks avaldise `error "Minu viga!"` interaktiivses keskkonnas väärtustada, tekib täitmisaegne viga, kus veateateks on “Minu viga!”.

Muutuja `error` rakendamise tulemus võib olla suvalise tüübi $\perp$, seetõttu on niimoodi võimalik täitmisaegset viga tekitada ühtmoodi sõltumata sellest, millist tüüpi tulemust kontekst nõuab.

Ülesandeid

78. Küsida interaktiivses keskkonnas muutuja `error` tüüp ja saada sellest aru.

79. Anda interaktiivse keskkonna käsurealt korrektne avaldis tüüpi `Int`, mille väärtustamisel lõpetatakse töö teie ette antud veateatega.

4. Andmete sõnekujud. Andmete esitusviis interaktiivse keskkonna poolt on määratud operaatori `show` väärtusega, mis kujutab endast funktsiooni, mis teisendab oma argumendi sõneks. Nimelt selleks, et vastust sõnekujul saada, lisavad interaktiivsed keskkonnad kõigile sisestatud avaldistele automaatselt operaatori `show` rakenduse.

Seega listide esitamine erisüntaksite kaudu tuleneb sellest, et operaator `show` selliselt on defineeritud.

Vastuse tõelisest struktuurist erinevat sõnekuju võib kohata teistegi tüüpide puhul. Standardteegis määratud sõneksteisendustel kehtib põhimõte, et andme sõnekuju on ise korrektne Haskelli kirjaviis tema esitamiseks.

Näiteks sõned pannakse jutumärkidesse ja asendatakse temas leiduvad erisümbolid nende langjoonkoodidega. Kodeerimist kasutatakse ka üksikute sümbolite korral.

Operaator `show` on defineeritud kõigi klassi Show kuuluvate tüüpide peal. Kõik tüübid aga pole sellised. Tähtsaim näide on funktsioonitüübid: funktsioonide sõneks teisendamine on standardteegis eeldefineerimata.

Vaikiv `show` lisamine interaktiivses keskkonnas viib selleni, et täiesti korrektse avaldise andmisel käsurealt tekib tüübiviga, kui see avaldis on tüüpi, mille jaoks on operaator `show` defineerimata, näiteks funktsioonitüüpi. Seega saab interaktiivse keskkonna käsurealt väärtustamiseks anda vaid täisargumenteeritud avaldisi.

Operaatori `print` rakendamisel mingile argumendile rakendatakse kõigepealt operaatorit `show` ja seejärel kirjutatakse saadud sõne standardväljundisse. Seega `print` on kasutatav ainult sellistel argumentidel, millel ka `show`.

Kui standardväljundisse soovitakse saata sõne, siis võib liigsete jutumärkide vältimiseks olla `print` asemel sobivam kasutada operaatoreid `putStr` ja `putStrLn`. Nende väärtuseks on funktsioonid, mis võtavad argumendiks sõne ja kirjutavad ta standardväljundisse, `putStrLn` seejuures lisab ka reavahetuse.

Ülesandeid

80. Sisestada interaktiivse keskkonna käsurealt avaldis, milles muutuja `show` on rakendatud mõnele arvule, sümbolile või tõeväärtusele ilmutatult. Mis vahe on saadavas vastuses võrreldes sellega, kui `show` on puudu, ja miks?

81. Loetleda järjest sümbolid avaldise `show 3`, avaldise `show (show 3)` ja avaldise `show (show (show 3))` väärtuseks olevas sõnes.

82. Anda interaktiivse keskkonna käsurealt täisargumenteerimata avaldisi ja saada aru järgnevast veateatest.

2.3.2 Aritmeetilise jada süntaks

Liste, mille elemendid moodustavad aritmeetilise progressiooni, saab moodustada omaette erisüntaksiga, aritmeetilise jada süntaksiga.

1. Süntaks. Aritmeetilise jada süntaksist on isegi neli üksteisega sarnanevat eraldi varianti, mis jagunevad kahekaupa vastavalt tõkkega ja tõkketa variantideks.

1.1. Tõkkega variandid. Üldine tõkestatud progressioon esitatakse kahe esimese elemendi ja tõkke kaudu kujul

$$\texttt{[}\mathfrak{a}\texttt{, }\mathfrak{b}\texttt{ .. }\mathfrak{c}\texttt{]}. \tag{5}$$

Kui $\mathfrak{a}$, $\mathfrak{b}$, $\mathfrak{c}$ väärtuseks on vastavalt arvud a, b, c, siis avaldise (5) väärtus on list, mille elemendid võetakse järjest aritmeetilisest jadast esimese elemendiga a vahega $b - a$ parajasti senikaua, kui jada liikmed pole arvteljel möödunud arvust c (täpsemalt, sattunud teisele poole c-d vaadates arvu $c - (b - a)$ poolt).

Näiteks avaldis `[1, 3, 5, 7, 9]` on samaväärne avaldisega `[1, 3 .. 9]`, samuti avaldisega `[1, 3 .. 10]`.

Kui progressioon on sammuga 1, võib avaldises (5) koma ja teise elemendi ära jätta.

Näiteks avaldise `[1 .. 10]` väärtus on list järjekorras arvudest 1 kuni 10.

Erijuhul, kui progressioon on sammuga 0 ehk teine element võrdub esimesega, saame lõpmatu listi, sest üle tõkke ei jõuta mitte kunagi.

Näiteks avaldise `[0, 0 .. 1]` väärtus on lõpmatu list, mille iga element on 0.

Ülesandeid

83. Sisestada aritmeetilise progressiooni süntaksiga avaldisi interaktiivse keskkonna käsurealt. Proovida nii positiivse kui negatiivse vahega progressioone ning kummalgi juhul nii esimesest elemendist suurema kui väiksema väärtusega tõkkega.

84. Arvutada kõigi 3-kohaliste 1-ga lõppevate arvude korrutis.

85. Arvutada suurim 2006-ga jaguv arv esimese 100000000 naturaalarvu seas, lastes selleks väärtustada aritmeetikatehteid mitte sisaldava avaldise.

86. Kirjutada interaktiivses keskkonnas avaldis, mille väärtuseks on list, milles on kasvavas järjestuses parajasti kõik positiivsed täisarvud kuni 1000-ni peale paaritute kolmekohaliste.

1.2. Tõkketa variandid. Et lõpmatud listid on praktilised, siis pole midagi imestada, et aritmeetilise progressiooni süntaksist leiduvad ka tõkketa variandid. Need saame senivaadeldutest, kui jätame tõket märkiva avaldise ära. Sellisel juhul on avaldise väärtuseks list, mille elemendid moodustavad terve aritmeetilise jada, mis on määratud esimese ja teise elemendiga või, kui ka teine element puudub, esimese elemendiga ja vahega 1.

Näiteks `[1, 3 .. ]` väärtus on list kõigi paaritute naturaalarvudega kasvavas järjestuses, `[1 .. ]` väärtus aga list kõigi positiivsete naturaalarvudega kasvavas järjestuses.

Ülesandeid

87. Kirjutada avaldis, mille väärtuseks on list, mille elementideks on kasvavas järjestuses kõik positiivsed täisarvud, mis annavad 7-ga jagades jäägi 6.

88. Kirjutada ühele reale (koos interaktiivse keskkonna käsurea viibaga alla 80 sümboli) mahtuv aritmeetikateheteta avaldis, mille väärtuseks on list, milles on 12 esimest 899979996999599949993999299919990999-ga jaguvat positiivset täisarvu kasvavas järjestuses.

89. Kirjutada interaktiivses keskkonnas avaldis, mille väärtus on lõpmatu list, mille iga element on kõigi naturaalarvude list.

90. Kirjutada interaktiivses keskkonnas avaldis, mille väärtus on kahes suunas lõpmatu liitmistabel listide listina, st listi nr n element nr m peab olema $n+m$, kus nummerdamine käib alates 0-st.

2. Aritmeetilised progressioonid mittearvulistel tüüpidel. Aritmeetilise progressiooni süntaksi kasutamiseks ei pea listide elemendid tingimata arvud olema. Nii saab koostada liste, mille elemenditüüp on suvaline tüüp klassist Enum.

2.1. Ümbertõlgendamine täisarvude kaudu. Tõlgendamaks aritmeetilise progressiooni avaldist mittearvuliste elementide puhul teisendatakse need elemendid lühikesteks täisarvudeks operaatoriga `fromEnum`, koostatakse list aritmeetilise progressiooniga nende baasil ja seejärel teisendatakse listi elemendid tagasi algsesse tüüpi operaatoriga `toEnum`.

Muutujad `fromEnum` ja `toEnum` tulevad moodulist Prelude ja omavad tähendust parajasti klassi Enum tüüpide jaoks. Nende väärtust võib mõista tüübiteisendusfunktsioonina või ka andmete kodeerimis- ja dekodeerimisfunktsioonina, kusjuures koodid on lühikesed täisarvud.

Samamoodi tähendasid juba varem vaadeldud operaatorid `ord` ja `chr` moodulist Data . Char vastavalt sümbolite kodeerimis- ja dekodeerimisfunktsiooni. Veel enamgi, sümbolitüüp kuulub klassi Enum, kusjuures `fromEnum` ja `toEnum` ongi väärtuselt võrdsed just nendesamade operaatoritega `ord` ja `chr`.

See näitab esiteks, et aritmeetilise progressiooni erisüntaksit saab kasutada ka sõnede esitamiseks. Teiseks, sümbolite järjestus vastab nende koodide kui arvude standardsele järjestusele. Kolmandaks tuleneb siit, et sümbolite koodide kasutamiseks pole moodulit Data . Char laadida vajagi.

Operaatori `toEnum` kasutamisel koodis võib olla vaja annoteerida tulemustüüp, kui kontekstist pole see tüüp üheselt selge. Tahtes leida sümbolit koodiga 33, tuleb kirjutada `toEnum 33 :: Char`, sest muidu süsteem ei tea, millisesse Enum-klassi tüüpi on vaja teisendada.

Ülesandeid

91. Küsida interaktiivses keskkonnas muutujate `fromEnum` ja `toEnum` tüübid ja saada neist aru.

92. Moodulit Data . Char sisse lugemata teha interaktiivse keskkonna abil kindlaks sümbol, mille kood on 39.

93. Teha kindlaks tõeväärtuste täisarvkoodid.

94. Arvutada interaktiivses keskkonnas kiiresti ladina tähestiku tähtede arv.

95. Kirjutada interaktiivse keskkonna käsurealt võimalikult lühike avaldis, mille väärtuseks oleks ühes sõnes ladina tähestik suurtähtedest, millele järgneks ladina tähestik väiketähtedest.

2.2. Aritmeetilised progressioonid lõplikel tüüpidel. Klassi Enum tüübid on tihti lõplikud, mistõttu vaid lõplik arv koode on kasutuses. Seetõttu võib tõkestamata aritmeetilise progressiooni süntaksi puhul juhtuda, et mingist kohast alates koodidele enam originaalandmeid ei vasta. Sellisel juhul sellel kohal list lõpetatakse. Seega tõkestamata aritmeetilise progressiooni süntaks võib väärtuseks ka lõpliku listi anda.

Näiteks `[False .. ]` väärtus on lõplik list elementidega False, True.

3. Ümberkirjutus operaatorite kaudu. Aritmeetilise jada süntaksi kirjutab süsteem automaatselt ümber moodulis Prelude defineeritud operaatorite `enumFromThenTo`, `enumFromTo`, `enumFromThen`, `enumFrom` kaudu vastavalt samaväärsustele

```
[a, b .. c]  ≡  enumFromThenTo a b c,
[a .. c]     ≡  enumFromTo a c,
[a, b .. ]   ≡  enumFromThen a b,
[a .. ]      ≡  enumFrom a.
```

Ülesandeid

96. Küsida interaktiivse keskkonna käsurealt muutujate `enumFromThenTo`, `enumFromTo`, `enumFromThen`, `enumFrom` tüübid ja saada neist aru.

3 Oma muutujate kasutamine

3.1 Lihtsad konstruktsioonid näidistega

Kõik eelnev on olnud vaid sissejuhatus, kus Haskellis programmeerimiseks kasutati vaid eeldefineeritud muutujaid ja konstruktoreid. (Erandiks olid polümorfsete avaldiste tüübid, milles sisalduvad lokaalsed tüübimuutujad, kuid neidki ei pidanud kasutamiseks defineerima.)

Uute muutujate defineerimiseks on vaja kasutada avaldistele tähenduselt vastanduvaid süntaktilisi üksusi — näidiseid. Nemad moodustavad käesoleva jaotise põhiteema. Enam ei saa mööda ka deklaratsioonidest.

Konstruktorite defineerimine on hoopis teistsugune asi ja jääb esialgu vaatluse alt välja. Vahe on selles, et muutuja defineerimisel seotakse muutuja juba põhimõtteliselt olemasoleva objektiga, kuid konstruktori defineerimine tähendab lisaks nimele ka uue objekti sissetoomist.

3.1.1 Avaldised ja näidised

Et selgitada näidiste ja deklaratsioonide olemust, vaatame võrdlusmomendi tekitamiseks ka avaldistele kõrgemalt vaatekohalt peale.

1. Avaldis. Avaldis esitab skeemi, kuidas üks objekt — avaldise väärtus — sõltub avaldise osade väärtustest.

Näiteks kui `x` väärtus on 2, `y` väärtus on 3 ja `+` väärtus on liitmine, siis avaldise `x + y` väärtus on $2 + 3$ ehk 5. Kui aga `x` väärtus on 5, `y` väärtus on 8 ja `+` väärtus on korrutamine, siis sama avaldise väärtus on $5 \cdot 8$ ehk 40. Niisiis avaldis `x + y` esitab skeemi, kuidas üks objekt sõltub muutujate `x`, `y` ja `+` väärtusest.

Võtame teise näitena avaldise `(a , b)`. Kui `a` väärtus on 2 ja `b` väärtus on 3,

siis avaldise `(a , b)` väärtus on $(2, 3)$. Kui `a` väärtus on 5 ja `b` väärtus on 8, siis sama avaldise väärtus on $(5, 8)$. Avaldis `(a , b)` esitab skeemi, kuidas üks objekt sõltub muutujate `a` ja `b` väärtustest.

Teisiti öeldes, avaldis esindab väärtuste laine liikumist "seest välja".

2. Näidis.

2.1. Näidise ülesanne. Näidis esitab skeemi, kuidas näidise osade väärtused ühest objektist — näidise väärtusest — sõltuvad.

Just selles mõttes on näidise ja avaldise mõisted duaalsed: näidis esindab väärtuste laine liikumist "väljast sisse".

Näiteks kui näidise `(a , b)` väärtus on $(2, 3)$, siis `a` väärtus on 2 ja `b` väärtus 3. Kui aga sama näidise väärtus on $(5, 8)$, siis `a` väärtus on 5 ja `b` väärtus on 8. Näidis `(a , b)` esitab skeemi, kuidas muutujate `a`, `b` väärtused sõltuvad paari väärtusest.

Nagu näha, võivad avaldis ja näidis ühtmoodi välja näha. Kontekst näitab, kas mõtteks on objekti väljendamine etteantud muutujate väärtuste kaudu või vastupidi — leida muutujate väärtused etteantud objekti järgi.

Seda, millised kirjutised üldjuhul kujutavad endast näidist, näeme edaspidi lihtsate deklaratsioonide peal. Kuid sõltumata näidise konstruktsioonist on muutujate kordumine näidises keelatud. Põhjus on selles, et niisugune näidis ei võimalda muutujate väärtusi üheselt määrata.

Näiteks `(a , a)` on näidisena illegaalne. Kui tema väärtus oleks $(2, 3)$, mida ei saa välistada, siis `a` väärtus peaks olema nii 2 kui ka 3.

2.2. Näidise sobitumine väärtusega. Et osade väärtusi määrata, peavad näidis ja tema väärtus struktuuri poolest klappima. Seda tingimust nimetatakse näidise sobitumiseks väärtusega. Näidis on niisiis otsekui objekti vorm, millel on selekteeriv roll: osa objekte vastavad vormile, osa mitte. Kui näidis oma väärtusega ei sobitu, siis tema kõigi väiksemate osade väärtuseks saab $\bot$.

Näiteks näidis `(a , b)` sobitub väärtustega $(2, 3)$ ja $(5, 8)$. Sama näidis aga ei sobitu väärtusega $\bot$, sest $\bot$ pole paarikujul, struktuur ei klapi. Sellisel juhul on `a` ja `b` väärtuseks $\bot$.

Sama näidis siiski sobitub väärtusega $(2, \bot)$ ja isegi väärtusega $(\bot, \bot)$. Näidise väärtuse $(2, \bot)$ korral saab `a` väärtuse 2, kuid `b` väärtuse $\bot$, väärtuse $(\bot, \bot)$ korral saavad mõlemad muutujad väärtuseks $\bot$.

Vaatame veel näidist `x : xs`. Tema sobitub kõigi mittetühjade listidega. Kui näidise väärtuseks satub list elementidega 1, 0 ehk `1:0:[]`, siis `x` väärtuseks on 1 ja `xs` väärtuseks `0:[]` ehk list elemendiga 0. Kui aga näidise väärtus on lõpmatu list `1:2:3:...`, siis `x` väärtus on 1 ja `xs` väärtus lõpmatu list `2:3:...`. Kui näidise `x : xs` väärtus on tühi list, siis näidis ei sobitu ning `x` ja `xs` väärtuseks jääb $\perp$.

2.3. Sobituselemendid. Avaldiste ja näidiste sellises käsitluses loetakse muutujate väärtused muutuvaks, kuid konstruktorite väärtused konstantseteks. Näiteks sulud ja koma tähendavad üheselt paarikonstruktorit, kuid + väärtus võis varieeruda. See näitab, et avaldise ja näidise elementaarosad, millest või milleni väärtuste laine liigub, ei tähenda siin igasugust süntaktiliselt jagamatut osa: konstruktorid, olgugi et jagamatud, jäävad mängust välja, nemad saavad oma väärtuse mujal.

Oleme tõele lähedal, kui ütleme, et elementaarosadeks on parajasti kõik muutujad, ja avaldiste puhul see täiesti nii ka on. Mis puutub näidistesse, siis Haskell võimaldab arvutusprotsessi sammude järjekorda varieerida, märkides keerulisemaid alamnäidiseid meelevaldselt elementaarseteks. Niisuguse näidise juures väärtuste omandamise laine areng sissepoole lõpeb. Näidise osi, mida loetakse elementaarselt sobituvaks, st mille juures väärtuste omandamise laine peatub, nimetame sõnastikuelementideks.

2.4. Näidise sobitamine avaldise vastu. Näidise väärtus, millega sobitumist tuleb kontrollida, on üldjuhul välja arvutamata. Tema kohal on mingi avaldis ning näidise sobitumise kontrollimisel väärtustatakse seda avaldist ainult niikaugele, et selguks, kas tema väärtuse ehitus vastab näidisele kuni sobituselementideni või võtab ta näidise struktuurile vastukäiva kuju. Seega isegi kontrolli lõppedes ei ole enamasti näidise väärtus teada.

Näiteks võiks näidise `x : xs` väärtust `1:2:3:...` reaalselt esindada avaldis `[1 .. ]`. Ka niisugusel juhul tehakse näidise sobitumine kindlaks, kuigi väärtuse väljaarvutamine on võimatu.

Kirjeldatud protsessi, mille osalised on näidis ja tema väärtust väljendav avaldis, nimetatakse näidise sobitamiseks. Kui avaldise väärtuse ehitus osutub näidisele vastavaks, siis loetakse sobitamine õnnestunuks ja näidise sobituselementidega seotakse vastavad alamavaldised selles avaldises, milleni algse avaldise väärtustamisel on jõutud.

Reaalselt on avaldise väärtustamise ja näidise sobitamise protsessid tihedalt põimunud: peaaegu iga avaldise väärtustamine nõuab näidiste sobitamisi ning peaaegu iga näidise sobitamine omakorda avaldiste väärtustamisi.

Märgime, et nii näidise sobitamise äsja esitatud üldkirjeldus kui ka edaspidised konkreetsemad selgitused arvutusprotsessi kohta on lihtsustatud, kuid selline viis asja mõista on programmeerijale üldiselt piisav ja ohutu. Haskelli keelekirjeldus tegelikult ei selgita põhimõtteliselt arvutusprotsessi detaile peaaegu üldse, vaid sätestab asjade tähendused ainult väärtuste tasemel, et arvutuse detailid võiksid tugevalt sõltuda realisatsioonist.

3.1.2 Lihtsad deklaratsioonid

1. Defineeriva deklaratsiooni üldkuju. Andmedeklaratsioonid, mis defineerivad uusi muutujaid, koosnevad vasakust ja paremast poolest. Lihtsaimal juhul on vasakuks pooleks üks näidis, parem pool aga koosneb võrdusmärgist ja avaldisest. Selline deklaratsioon on niisiis kujul

$$\mathfrak{p} \texttt{ = } \mathfrak{e}. \tag{6}$$

Kõige abstraktsemalt on deklaratsiooni (6) mõte omistada näidise $\mathfrak{p}$ väärtuseks avaldise $\mathfrak{e}$ väärtus. See loob potentsiaali näidises $\mathfrak{p}$ esinevate muutujate väärtuste määramiseks: avaldise muutujad määravad avaldise väärtuse, mis on deklaratsiooni põhjal sama mis näidise väärtus, ning see võib määrata näidise muutujate väärtused. Kuidas see konkreetselt realiseerub, on iseküsimus, mida vaatleme hiljem. Vaatame kõigepealt sellise deklaratsiooni näitel läbi tähtsamad näidiseliigid.

2. Näidiste liigid.

2.1. Muutuja. Näidise võib moodustada üksikust muutujast. Muutuja ainus sobituselement on tema ise.

Näiteks deklaratsioon

```
alus
  = 10
```
(7)

defineerib muutuja `alus` väärtuseks 10.

Kujul (6) oli ka meie esimene kirjutatud deklaratsioon (1). Seal oli vasakuks pooleks muutuja `main` ja paremaks pooleks avaldis `print (2 + (-3))`.

Ülesandeid

97. Kirjutada deklaratsioon (7) oma koodifaili. Testida defineeritud muutujat interaktiivses keskkonnas.

98. Lisada deklaratsioon, mis defineerib muutuja `e` väärtuseks arvu e ujukomalise lähendi.

2.2. Konstruktoriga näidis. Teine põhiline liik näidiseid saadakse konstruktori abil, kus argumentide kohale pannakse omakorda näidised.

Seni vaatlesime kaht sellist näidist: `(a , b)`, mis koostatud muutujatest `a` ja `b` paarikonstruktori abil, ja `x : xs`, mis koostatud muutujatest `x` ja `xs` listikonstruktori `:` abil.

Siia kuuluvad ka näidised, mis koosnevad paljast konstruktorist, mille väärtus pole funktsioon, nagu näiteks `[]`, `True`, `False`, `Nothing`.

Konstruktori abil näidise koostamine käib nagu sama konstruktori täisargumenteeritud rakendamine avaldistele. Muuhulgas on infiksoperaatorite prioriteedid ja assotsiatiivsus samad mis avaldistes, samuti on võimalik kasutada muutujate infiks- ja prefikskuju. Täisargumenteerituse nõue tagab, et konstruktoriga koostatud näidis ei ole funktsioonitüüpi.

Näiteks näidis `x : xs` on samaväärne näidisega `(:) x xs`.

Näidise `(:) x` aga lükkab Haskell tagasi, sest ta on funktsioonitüüpi, ta vajab veel üht argumenti.

Konstruktoriga koostatud näidise sobituselemendid on parajasti kõik konstruktori argumentide sobituselemendid.

Konstruktori olemusest tulenevalt on konstruktori argumentide väärtused alati konstruktori rakendamise tulemusest üheselt välja loetavad. Seega küsimuse konstruktoriga ehitatud näidise sobitumisest oma väärtusega saab lahendada järgmiselt. Kui väärtus ei ole moodustatud sama konstruktoriga, siis näidis ei sobitu temaga, sest struktuur ei klapi. Kui väärtus on moodustatud sama konstruktoriga, siis sobitub kogu näidis väärtusega parajasti siis, kui kõik konstruktori argumentnäidised sobituvad konstruktori vastavate argumentidega väärtuses, ning sealt saame ka sobituselementide väärtused.

Kui konstruktoril argumendid puuduvad, siis puuduvad näidisel ka sobituselemendid. Näidise sobitamine sellisel juhul taandub kontrollile, kas konstruktor ja avaldis on väärtuselt võrdsed.

Lihtne näide deklaratsioonist kujul (6), mille vasak pool on konstruktoriga ehitatud näidis, on

```
(x , y)
  = (5 , 0.2).
```
(8)

Kuna näidise ja avaldise struktuurid sobivad, siis see deklaratsioon määrab `x` väärtuseks 5 ja `y` väärtuseks 0.2.

Deklaratsioon

```
p : ps
  = 1 : 3 : 4 : []
```
(9)

annab muutujale `p` väärtuseks 1 ja muutujale `ps` väärtuseks lõpliku listi elementidega 3, 4. Soovi korral võime rohkem elemente algusest muutujatega välja tuua, näiteks deklaratsioonist

```
p : q : qs
  = 1 : 3 : 4 : []
```
(10)

saaks `p` väärtuse 1, `q` väärtuse 3 ja `qs` väärtuseks lõpliku listi elemendiga 4.

Muidugi saame deklaratsioonid (9), (10) samaväärselt ümber kirjutada, kasutades paremates pooltes listide erisüntaksit. Saame vastavalt

```
p : ps
  = [1, 3, 4]'

p : q : qs
  = [1, 3, 4]'
```

Kuna sõned on sümbolite listid, on korrektne ka näiteks deklaratsioon

```
c : cs
  = "Hei!"'
```
(11)

sest vasakul olev näidis sobitub iga mittetühja listiga, muuhulgas sõnega “Hei!”. Selle deklaratsiooni tulemusel saab `c` väärtuseks H-tähe ning `cs` sõne “ei!”.

Ülesandeid

99. Millised näidistest

    ```
    (:), (:) x xs xss, (x : xs) : xss, [],
    Just, Just a, Just sin, Just (+), Just Nothing
    ```

    on korrektsed?

100. Defineerida ühe deklaratsiooniga kaks muutujat, mille väärtuseks on vastavalt Teie ees- ja perekonnanimi sõnena. Anda interaktiivse keskkonna käsurealt neid muutujaid sisaldav avaldis, mille väärtuseks oleks Teie täisnimi eesnimega ees. Seejärel anda avaldis, mille väärtuseks oleks Teie ametlik täisnimi perekonnanimega ees. Tulemused peavad vastama eesti keele reeglitele.

101. Kirjutada oma moodulisse deklaratsioon, mis defineerib muutujate `q` ja `r` väärtuseks vastavalt arvujärjendi $0.501, 0.502, \ldots, 1.499$ arvude korrutise täis- ja murdosa. Anda interaktiivse keskkonna käsurealt avaldis, mille väärtuseks on korrektne eestikeelne lause, mis ütleb, mis on selle korrutise täisosa ja mis on murdosa.

102. Modifitseerida deklaratsiooni (11) paremat poolt selliselt, et deklaratsiooni tulemusel saaks `cs` väärtuseks tühja sõne.

2.3. Erisüntaksiga näidised. Ka näidistes lubab Haskell kasutada listide erisüntaksit elementide loeteluga. Näiteks on võimalik muutujad `x`, `y`, `z` defineerida deklaratsiooniga

```
[x, y, z]
  = [1, 3, 4].
```

Põhimõtteliselt on näidisena kasutatavad ka sõnekonstandid, samuti sümbol- ja arvkonstandid.

Ülesandeid

103. Kas saab kirjutada deklaratsiooni (9) vasaku poole samaväärselt ümber elementide loetelu kujul?

2.4. Ehknäidis. Näidise moodustamisel konstruktoriga viitavad konstruktori argumendid struktuuri piirkondadele, mis omavahel ei kattu. Seega ka erinevate muutujate tähistatavad osad omavahel ei kattu.

Kuid saab kirjutada ka nii, et näidise muutujate väärtustest üks teist sisaldaks. Seda võimaldab nn **ehknäidis** kujul

```
ɾ@ ƥ,
```

kus `ɾ` on muutuja ja `ƥ` näidis. Sümbol `@` tähendab põhimõtteliselt näidiste võrdust, st kui näidise `ɾ@ ƥ` väärtus on x, siis on `ɾ` väärtus x ja `ƥ` väärtus x. Tulemusena on näidise `ƥ` muutujate väärtused muutuja `ɾ` väärtuse osad.

Sümboli valik tuleb ingliskeelsest sõnast "*as*". Eesti keeles võib näidist kujul `ɾ@ ƥ` lugeda "`ɾ` ehk `ƥ`".

Ehknäidise `ɾ@ ƥ` sobituselemendid on parajasti `ƥ` sobituselemendid ja `ɾ`.

Näiteks võiksime täiendada deklaratsiooni (8) vasakut poolt selliselt, et lisaks muutujate `x` ja `y` väärtuse määramisele kirjutataks kogu paari väärtus muutujasse `z`:

```
z@ (x , y)
  = (5 , 0.2).
```
(12)

Ehknäidise konstrueerimisel tuleb arvestada, et `@` seob tugevamini kõigist infiksoperaatoritest ja isegi funktsioonirakendamisest (piltlikult öeldes on `@` prioriteediga 11).

Vaatleme veel deklaratsiooni

```
t : ts@ (u : us)
  = "string"
```
. (13)

Kuna `@` on kõigist infikssetest seostest tugevaim, siis on vasaku poole näidis sama mis `t : (ts@ (u : us))`. Seega deklaratsiooni (13) tulemusena saab muutuja `t` väärtuseks s-tähe, näidis `ts@ (u : us)` aga sõne "tring". Järelikult nii muutuja `ts` kui näidise `u : us` väärtuseks saab "tring". Siit tulenevalt saab muutuja `u` väärtuseks t-tähe ja muutuja `us` väärtuseks sõna "ring".

Ülesandeid

104. Kasutades ülesande 98 lahendust, kirjutada võimalikult lühike deklaratsioon, mis defineerib muutuja `eee` väärtuseks arvu e^{e^e} ujukomalise lähendi ja muutuja `eeeList` väärtuseks listi, mille ainsaks elemendiks on seesama väärtus.

105. Kirjutada võimalikult lühike deklaratsioon, mis defineerib muutujate `c1`, `c2`, `c3` väärtuseks vastavalt 1, 2, 3 ning muutuja `d` väärtuseks $(2, 3)$.

2.5. Jokker. Veel üks näidiseliik on **jokker**, mis kirjutatakse üksiku alakriipsuna. Jokker on sobituselementideta näidis, mis sobitub iga väärtusega. Jokkerit võib lugeda väljendiga "ükskõik mis".

Praktikas märgitakse jokkeriga ebaolulisi objekte; väärtustamisel unustatakse selline objekt ära, kuna ühtki tema osa ei seota. Mittekasutatavate muutujate asendamine jokkeriga hoiab kokku arvuti mälu ja säästab ka koodi uurivat inimest nende muutujate tähenduse meelespidamisest.

Näiteks on täiesti legaalne deklaratsioon

```
_ = 15.
```

Selline deklaratsioon on muidugi täiesti kasutu.

Küll aga on jokkerit mõnikord mõistlik kasutada suurema näidise koosseisus. Vaatleme näiteks deklaratsiooni (12), mis defineeris muutujad `x`, `y`, `z`. Oletame, et meil on vaja kasutada muutujaid `x` väärtusega 5 ja `z` väärtusega $(5, 0.2)$, kuid `y` väärtusega 0.2 tarvis ei lähe. Siis tasub deklaratsioonis (12) asendada `y` jokkeriga, kirjutades

```
z@ (x , _)
  = (5 , 0.2).
```

Ülesandeid

106. Lahendada ülesanne 104 lisatingimusel, et listide erisüntakseid näidistes ei kasuta. Lisamuutujaid mitte defineerida.

107. Kirjutada võimalikult lühike deklaratsioon, mis defineerib muutuja `ae` väärtuseks sõne "ARKTIKA-EKSPEDITSIOONI MÄLESTUSED" ja `ae'` väärtuseks sõne "ANTARKTIKA-EKSPEDITSIOONI MÄLESTUSED" ning muid muutujaid ei defineeri.

2.6. Laisk näidis. Viimane liik näidiseid, mida me käsitleme, on nn laisad näidised kujul

```
~p,
```
(14)

kus `p` on suvaline näidis.

Näidise väärtust lisatav tilde ei muuda: `p` väärtus näidises (14) võrdub kogu näidise väärtusega. Laisk näidis on aga võimalus suuremat näidist "musta kasti panna" ehk elementaarseks kuulutada: sõltumata `p` ehitusest on näidisel (14) parajasti üks sobituselement ja see on `p`.

Laisk näidis sobitub iga väärtusega, sest sõltumata väärtusest seotakse see näidisega `p`, mille struktuuri ei uurita. Seepärast kasutatakse terminit laisk tihti ka üldisemalt igasuguse näidise jaoks, mis sobitub suvalise väärtusega, nagu näiteks muutujad ja jokker.

Vaatleme näiteks deklaratsiooni

```
p : q : qs
  = 1 : [].
```
(15)

Siin vasak pool ei sobitu parema poole väärtusega, sest alamnäidis `q : qs` ei sobitu tühja listiga, struktuur ei klapi.

Kui aga teha probleeme tekitanud argument laisaks näidiseks, saame deklaratsiooni

```
p : ~(q : qs)
  = 1 : []   .
```
(16)

Nüüd vasak pool sobitub parema poole väärtusega, sest `q : qs` on sobituselement, tema sobitumine oma väärtusega pole vajalik. Tulemusena saab `p` väärtuseks 1.

Küsides muutuja `p` väärtust deklaratsioonist (15) tekib näidisesobituse viga, kuid deklaratsiooni (16) puhul seda ei teki. Deklaratsioon (16) annab vea alles siis, kui küsida muutuja `q` või muutuja `qs` väärtust. Muutuja `p` väärtuse erinevust deklaratsioonide (15) ja (16) korral saab kergesti kontrollida interaktiivses keskkonnas.

Laiska näidist kasutatakse reaalselt täitmisaegsete vigade ennetamiseks. Praktilisi näiteid praeguses õppejärgus tuua pole võimalik, neid näeme hiljem.

3. Arvutus defineeriva deklaratsiooni järgi. Oleme nüüd valmis saama ettekujutust sellest, mis masina sees võiks toimuda, kui eksisteerib deklaratsioon kujul (6). Protsess on põhimõtteliselt järgmine.

Kuni ühegi vaadeldava deklaratsiooni poolt defineeritava muutuja väärtust vaja ei lähe, seda deklaratsiooni ei puudutata.

Seetõttu aktsepteerib Haskell vaikides taolisi tüübikorrektseid, kuid absurdseid deklaratsioone nagu `True = False`.

Oletame nüüd, et vajatakse muutuja $\mathfrak{x}$ väärtust, kus $\mathfrak{x}$ esineb kujul (6) oleva deklaratsiooni vasakus pooles $\mathfrak{p}$. Sellisel juhul sobitatakse näidist $\mathfrak{p}$ avaldise $\mathfrak{e}$ vastu. See tähendab, et avaldis $\mathfrak{e}$ väärtustatakse mingile kujule $\overline{\mathfrak{e}}$, millest on $\mathfrak{p}$ sobitumine või mittesobitumine tema väärtusega kujude võrdluses näha.

Mittesobitumise korral lõpetatakse töö täitmisaegse veaga. Sobitumise korral seotakse näidise $\mathfrak{p}$ kõik sobituselemendid avaldise $\overline{\mathfrak{e}}$ vastavate alamavaldistega (need võivad veel olla väärtustamata). Kui $\mathfrak{x}$ oli üks sobituselementidest, siis saab temaga seotud avaldisest hakata tema väärtust arvutama ja deklaratsioon (6) on end selleks korraks ammendanud. Kui $\mathfrak{x}$ polnud sobituselement, siis otsitakse välja sobituselement $\mathfrak{p}'$, kus $\mathfrak{x}$ esineb; olgu $\mathfrak{e}'$ temaga seotud avaldis. Edasi toimitakse nii, nagu oleks kirjutatud deklaratsioon kujul (6), kus vasak ja parem pool on vastavalt $\mathfrak{p}'$ ja $\mathfrak{e}'$.

Illustreerime seda protsessi deklaratsiooni (8) peal. Oletame, et on vaja muutuja `x` väärtust. Kuna parema poole avaldis on paarikujul, on vasaku poole näidise sobitumine tema väärtusega ilma midagi tegemata selge. Muutuja `x` seotakse avaldisega `5` ja muutuja `y` avaldisega `0.2`. Edasi võib deklaratsiooni (8) kõrvale panna, sest `x` väärtuse arvutamiseks sobiv avaldis on käes. Protsessi kõrvalt sai omale väärtuse ka muutuja `y`, nii et kui edaspidi selle muutuja väärtust vaja on, siis ei pea deklaratsiooni (8) enam kasutama.

Põhimõtteliselt samamoodi käivad asjad deklaratsiooni (10) puhul. Ükskõik, kas on vaja muutuja `p`, muutuja `q` või muutuja `qs` väärtust, seotakse nad kõik vastavalt avaldistega `1`, `3` ja `4 : []`.

Vaatame nüüd deklaratsiooni

```
p : ~(q : qs)
  = 1 : 3 : 4 : [].
```

Kui on vaja muutuja `p` väärtust, siis seotakse sellest deklaratsioonist `p` avaldisega `1` ja näidis `q : qs` avaldisega `3 : 4 : []`. Kuna `p` väärtuse arvutamiseks vajalik avaldis on käes, siis pannakse deklaratsioon kohe kõrvale, `q` ja `qs` jäävad sidumata.

Kui aga oleks küsitud muutuja `q` väärtust, siis tulnuks sobitada veel näidist `q : qs` avaldise `3 : 4 : []` vastu, mille tulemusel oleks `q` seotud avaldisega `3`.

Ülesandeid

108. Kirjutada deklaratsioon (15) oma koodifaili ja tekitada selle kaudu interaktiivses keskkonnas näidisesobituse viga.

109. Olgu meil deklaratsioonid

```
p : q : qs
  = [1]   ’

p : ~(q : qs)
  = [1 : 3 : []]’

[p : ps]
  = "Has" : "kell" : []’
```

Iga deklaratsiooni kohta otsustada, (1) kas ta on tüübikorrektne, (2) kui jah, siis mis saab muutuja `p` väärtuseks.

110. Demonstreerida arvutis, et muutuja, jokkeri ja laisa näidisega sobitub ka $\bot$.

111. Olgu meil deklaratsioon

```
~(x , (a , b))
  = (1 , error "VIGA")’
```

Selgitada, miks muutuja `x` väärtustamine lõpeb veateatega. Tõsta antud deklaratsioonis 1 sümbol ümber nii, et `x` väärtuseks saaks 1.

4. Tüübisignatuurid. Haskellis saab kirjutada tüübisignatuure — eraldi deklaratsioone, mis näitavad defineeritavate muutujate tüüpi.

Tüübisignatuur on kujul

$$\mathfrak{x}_1\texttt{, } \ldots\texttt{, } \mathfrak{x}_l \texttt{ :: } \mathfrak{t},$$

kus $\mathfrak{x}_1, \ldots, \mathfrak{x}_l$ on muutujad ja $\mathfrak{t}$ tüübiavaldis. Tulemusena loetakse muutujate $\mathfrak{x}_1, \ldots, \mathfrak{x}_l$ tüübiks $\mathfrak{t}$.

Tüübisignatuur muutujat ei defineeri, tüübisignatuuri vasakus pooles olevad muutujad peavad olema defineeritud teiste deklaratsioonidega. Muutuja tüübisignatuuri paremas pooles olev tüüp peab olema võrdne tüübiga, mille see muutuja saaks oma definitsioonist, või seda tüüpi täpsustama, vastasel korral tekib tüübiviga.

Näiteks deklaratsioonile (7), mis defineerib muutuja `alus` väärtuseks 10, võib lisada tüübisignatuuri

```
alus
  :: (Num a) => a
```
(17)

Kui tahame kitsendada muutuja `alus` tüübiks `Integer`, võib signatuuri (17) asemel kirjutada

```
alus
  :: Integer
```

Enamasti on tüübisignatuuri vaja just tüübi kitsendamiseks, kuna süsteem suudab üldjuhul definitsiooni järgi muutujate tüübid tuletada. Mõnikord siiski vajab süsteem tüübituletamisel abi, mille saab anda tüübisignatuuri lisamisega. Tüübisignatuuride kirjutamine on mõttekas igal juhul, kuna see hõlbustab koodist arusaamist.

Ülesandeid

112. Lisada ülesande 98 lahendusele tüübisignatuur mitmel viisil, varieerides tüübipere suurust.

5. Tüübisünonüümide defineerimine. Kuigi seni oleme kasutanud vaid andmenäidiseid, on näidistest mõtet rääkida ka tüüpide tasemel. Peaaegu kujul (6) deklaratsiooniga — ette tuleb vaid lisada võtmesõna **`type`** — saab defineerida globaalseid tüübimuutujaid (tüübisünonüüme).

Näiteks võime nii defineerida tüübimuutuja `Rida`, mille väärtuseks on täisarvude listide tüüp. Deklaratsiooniks tuleb

```
type Rida
  = [Integer]
```

Nüüd saab koostada tüübiannotatsiooniga avaldisi nagu `[1 .. ] :: Rida`.

Siiski on tüübitasemel võrreldes andmetasemega tugevad kitsendused. Tüübinäidisena tohib kasutada ainult paljast muutujat; tüübisünonüümi definitsiooni parema poole tüübiavaldis peab olema tüübikontekstita.

Ülesandeid

113. Otsida Hugsi teegist üles tüübimuutuja `String` definitsioon ja saada sellest aru.

3.1.3 Lambdaavaldised

Teine tüüpiline koht, kus näidised figureerivad, on funktsioonide kirjeldused, seal täidavad näidised formaalsete parameetrite aset.

1. Nimetud funktsioonid. Lihtsaimad süntaktilised konstruktsioonid, kus näidised märgivad kirjeldatava funktsiooni argumente, on nn lambdaavaldised, mis kirjeldavad funktsiooni ilma talle nime andmata.

Lambdaavaldis on kujul

```
\ p -> e,                                                    (18)
```

kus `p` on näidis ja `e` avaldis. Noole moodi kombinatsioon `->` nagu varem võrdusmärkki jagab kirjutise vasakuks ja paremaks pooleks. Sümbolit `\` selle konstruktsiooni kontekstis nimetatakse lambdaks.

Avaldise (18) väärtus on funktsioon, mis igal oma argumendil x annab tulemuseks avaldise `e` väärtuse olukorras, kus näidise `p` väärtus on x ja näidise `p` muutujad on avaldises `e` sama väärtusega nagu näidises. Erandiks on argumendid, millega näidis ei sobitu — neil on funktsiooni väärtus $\bot$.

Näiteks avaldise `\ x -> x * x` väärtus on funktsioon, mis igal oma argumendil x annab tulemuseks avaldise `x * x` väärtuse eeldusel, et `x` väärtus on x, ehk parajasti x^2. Öeldes lühidalt, `\ x -> x * x` väärtus on ruutfunktsioon. Seega on korrektne näiteks rakendamine `(\ x -> x * x) (1 + 2)`, selle avaldise väärtus on 9.

Avaldise `\ (x , _) -> x` väärtus on funktsioon, mis igal oma argumendil $(a\,,b)$ annab tulemuseks muutuja `x` väärtuse eeldusel, et näidise `(x , _)` väärtus on $(a\,,b)$, niisiis a. Öeldes lühidalt, `\ (x , _) -> x` väärtus on paari projektsioon esimesele komponendile, sama mis eeldefineeritud muutujal `fst`. Jokkeri asemel võiks olla ka mõni `x`-st erinev muutuja, aga kuna seda muutujat kuskil ei kasutata, on jokker sobivam.

Niisiis lambdaavaldiste puhul on väärtuste liikumise suund vastupidine võrreldes deklaratsiooniga (6). Kui deklaratsioonis (6) liigub väärtus avaldisest näidisesse, siis lambdaavaldises liiguvad muutujate väärtused näidisest avaldisse, näidisesse liigub aga argumendi väärtus.

Lambdaavaldise vasakus pooles esinevad muutujad on lokaalsed, nähtavad ainult selle lambdaavaldise piires. Lokaalsete muutujate nimed tohivad globaalsete muutujate nimedega kokku langeda; sellisel puhul pole vastavad globaalsed muutujad lokaalse muutuja nähtavuspiirkonnas kasutatavad.

Näiteks `x` avaldises `\ x -> x * x` on lokaalne muutuja, tema nähtavus piirdub ainult selle lambdaavaldisega.

Avaldise `\ sin -> sin * sin` väärtus on samuti ruutfunktsioon, kuigi `sin` on eeldefineeritud muutuja. Ta ei tähenda selles avaldises siinust.

Ülesandeid

114. Testida ruutfunktsiooni interaktiivses keskkonnas.

115. Kuidas testida, et `\ (x , _) -> x` ja `fst` on sama väärtusega?

116. Kirjutada lambdaavaldis, mille väärtus on funktsioon, mis võtab argumendiks arvupaari ja annab väärtuseks komponentide vahe absoluutväärtuse.

117. Kirjutada lambdaavaldised, mille väärtused on vastavalt võrdsed eeldefineeritud muutujate `head` ja `tail` väärtustega, seejuures neid muutujaid endid kasutamata. Kus võimalik, kasutada jokkerit.

118. Kirjutada lambdaavaldis, mille väärtus on funktsioon, mis võtab argumendiks listi ning annab välja tema teise elemendi, kui see leidub, vastasel korral lõpetab täitmisaegse veaga. Kasutada võimalikult vähe muutujaid.

119. Kirjutada võimalikult lühike avaldis, mille väärtus on funktsioon, mis võtab argumendiks paari ja annab välja paari, mille esimene komponent on argumentpaar ja teine komponent on argumentpaar vahetatud komponentidega.

120. Kirjutada avaldis, mille väärtus on funktsioon, mis võtab argumendiks arvu ja annab tulemuseks lõpmatu listi, mille elementideks on kõik selle arvu naturaalarvkordsed kordaja kasvamise järjestuses.

1.1. Väärtustamine. Lambdaavaldist saab väärtustada ainult koos argumendiga. Avaldise `(\ 𝔭 -> 𝔢) 𝔞` väärtustamine algab näidise `𝔭` sobitamisega avaldise `𝔞` vastu. Sobitumise korral seotakse näidise sobituselemendid nagu ikka näidise sobitamisel, kogu lambdaavaldis aga kirjutatakse ümber lihtsalt avaldiseks `𝔢`, mida siis vajadusel edasi väärtustatakse. Mittesobitumise korral antakse täitmisaegne viga.

Võib paista, et see tähendab sama mis väärtustada avaldis `𝔢` olukorras, kus on antud deklaratsioon `𝔭 = 𝔞`. Päris nii see ei ole, sest selles olukorras alustataks avaldise `𝔢` väärtustamisega, mitte näidise sobitamisega, ja kui näidise `𝔭` muutujaid vaja ei läheks, siis jääks näidis üldse sobitamata. Kirjeldatud olukorraga ühtmoodi käiks avaldise `(\ ~𝔭 -> 𝔢) 𝔞` väärtustamine.

Vaatleme näitena avaldise `(\ x -> x * x) (1 + 2)` väärtustamist. Kuna näidis `x` on muutuja, siis ta sobitub suvalise väärtusega, nii et argumenti `1 + 2`

sellel etapil ei väärtustata, muutuja `x` seotakse avaldisega `1 + 2`. Siis kirjutatakse algne avaldis ümber avaldiseks `x * x`. Nüüd on juba näha, et edasine väärtustamine annab tulemuseks `9`, aga selles faasis ei mängi lambdaavaldis enam rolli.

Avaldise

```
(\ (x : xs) -> const 5 x) [1 .. ]
```
(19)

väärtustamisel sobitatakse näidis `x : xs` avaldise `[1 .. ]` vastu. Tulemusena seotakse muutuja `x` avaldisega `1` ja muutuja `xs` mingi avaldisega, mille väärtus on lõpmatu list täisarvudega alates 2-st. Kogu avaldis kirjutatakse ümber avaldiseks `const 5 x`, mille väärtustamine annab muutujat `x` lugemata tulemuseks `5`.

Kui avaldises (19) panna `[1 .. ]` asemele `[]`, siis näidis ei sobitu ja tekib täitmisaegne viga, ehkki näidise muutujaid `x` ja `xs` tegelikult vaja ei ole. Kui aga lisaks muuta näidis lambdaavaldises laisaks, st tekib avaldis

```
(\ ~(x : xs) -> const 5 x) [],
```
(20)

siis näidis sobitub. Avaldis (20) kirjutatakse ümber avaldiseks `const 5 x`, mis omakorda annab ilma `x` vastu huvi tundmata välja `5`.

Ülesandeid

121. Selgitada detailides avaldise `(\ (x , _) -> x) (1 + 1 , 2 - 5)` väärtustusprotsessi.

122. Asendada avaldises (20) muutuja `const` sellise muutujaga, et tulemuseks oleva avaldise väärtustamine lõpetab täitmisaegse veaga.

2. Nimetud karritatud funktsioonid. Lambdaavaldisega on võimalik üles kirjutada ka karritatud funktsioone. Selleks peab parem pool olema funktsioonitüüpi avaldis, näiteks omakorda lambdaavaldis.

Karritatud funktsiooni, mis leiab kahe arvu aritmeetilise keskmise, kirjeldab avaldis

```
\ a -> \ b -> (a + b) / 2.
```

Analoogse funktsiooni kolme argumendi korral kirjeldab

```
\ a -> \ b -> \ c -> (a + b + c) / 3.
```

Karritatud funktsioonide jaoks on olemas ka kirjutamist lihtsustav erisüntaks, mis lubab korduvad lambdad ja nooled kaotada, jättes alles ainult esimese lambda ja viimase noole ning kirjutades kõik argumendid lihtsalt üksteise järele neid vajadusel tühisümbolitega eraldades.

Vaadeldud aritmeetilise keskmise funktsioonid oleksid selle süntaksiga vastavalt

```
\ a b -> (a + b) / 2,
```

```
\ a b c -> (a + b + c) / 3.
```

Kehtib nõue, et sama lambda alla kogutud näidiste reas ei tohi ükski muutuja esineda korduvalt (muidu nõutakse seda vaid igas näidises eraldi).

Ülesandeid

123. Kirjutada lambdaavaldis, mille väärtus on funktsioon, mille saame operaatori `snd` väärtuseks oleva funktsiooni karritamisel.

124. Kirjutada avaldis, mille väärtus on ülesandes 116 kirjeldatud funktsiooni karritamise tulemus. Testida interaktiivses keskkonnas.

125. Kirjutada lambdaavaldis, mille väärtus on karritatud funktsioon, mis võtab argumendiks kaks listi: kui need mõlemad on kaheelemendilised, siis annab välja sellise 2×2 maatriksi determinandi, mille ridades on parajasti argumentlistide elemendid, vastasel korral lõpetab täitmisaegse veaga.

3. Funktsiooni andmine muutuja väärtuseks. Pannes deklaratsiooni (6) paremaks pooleks lambdaavaldise, saame defineerida muutuja väärtuseks endakoostatud funktsioone.

Näiteks võime anda ruutfunktsiooni muutuja `sqr` väärtuseks deklaratsiooniga

```
sqr
  = \ x -> x * x'
```
(21)

kahe ja kolme arvu aritmeetilise keskmise operatsioonid vastavalt muutujate `am2` ja `am3` väärtuseks aga deklaratsioonidega

```
am2
  = \ a b -> (a + b) / 2'
```
(22)

```
am3
  = \ a b c -> (a + b + c) / 3.
```
(23)

Muutujad, mille väärtuseks defineeritakse funktsioon, on sellega kohe kasutatavad prefiksselt, karritatud funktsiooni puhul ka infiksselt.

Deklaratsioonide (22) ja (23) olemasolul on avaldiste `am2 3 5` ja `3 ‘am2‘ 5` väärtus 4, avaldiste `am3 1 6 8` ja `(1 ‘am3‘ 6) 8` väärtus 5.

Kui on ette näha, et uut muutujat hakatakse oma argumentidele rakendama peamiselt infiksselt, on mõistlik muutujanimi koostada sümbolitest reas (3).

Näiteks deklaratsioon

```
(//)
  = \ a x -> fromInteger a + 1 / x
```
(24)

annab muutujale `//` väärtuseks karritatud funktsiooni, mis võtab argumendiks täisarvu a ja murdarvu x ja annab tulemuseks arvu $a + \frac{1}{x}$. Muutuja `fromInteger` väärtuseks on tüübiteisendus, mis viib täisarvu üle suvalisse vajalikku arvutüüpi suuruselt samaks arvuks; deklaratsioonis (24) on `fromInteger` vajalik selleks, et `+` argumendid oleksid üht tüüpi.

3.1. Infikskuju atribuutide seade. Värskelt kasutusele võetud nime infikskuju prioriteet ja assotsiatiivsus võrduvad vaikeväärtustega, kui neid pole ümber defineeritud **infiksdeklaratsiooniga**. Infiksdeklaratsiooniks sobib selline rida, mille esitavad infiksoperaatorite prioriteedi-assotsiatiivsuse kohta kasutajale interaktiivsed keskkonnad. Kuid sama infiksdeklaratsiooniga saab sama prioriteeti ja assotsiatiivsust seada mitme nime jaoks, selleks tuleb nime asemel kirjutada komadega eraldatud nimede loend.

Näiteks meie aritmeetilist keskmist arvutavate infiksoperaatorite assotsiatiivsuse ja prioriteedi muudab sarnaseks korrutus- ja jagamismärgiga deklaratsioon

```
infixl 7 `am2`, `am3`.
```

Infiksdeklaratsioon nime mujal infiksselt kasutama ei kohusta.

Ülesandeid

126. Lisada definitsioonile (24) infiksdeklaratsioon, mis annab infiksoperaatorile `//` jagamisega võrdse prioriteedi ja mõistliku assotsiatiivsuse.

3.2 Hargnemiskonstruktsioonid

Tutvume järgnevalt nelja süntaktilise konstruktsiooniga, mis võimaldavad erinevate juhtude läbivaatust.

Kaks esimest süntaktilist konstruktsiooni moodustavad avaldisi, teised kaks aga juba suuremaid süntaktilisi üksusi. Vaatleme neid kõiki uute muutujate definitsioonide koosseisus. Ühelt poolt on hargnemisega süntaktilised

üksused enamasti liiga pikad, et neid interaktiivses keskkonnas käsurealt sisestada, teiselt poolt on hargnemine mõttekas ainult siis, kui ta toimub mingi tundmatu objekti põhjal, milleks tüüpiliselt on defineeritava funktsioonitüüpi muutuja argument.

3.2.1 Tingimusavaldis

1. *if*-konstruktsioon Haskellis. Lihtsaim hargnemisega konstruktsioon on tingimusavaldis kujul

```
if 𝔟 then 𝔢1 else 𝔢2,
```
(25)

kus $\mathfrak{b}$, $\mathfrak{e}_1$ ja $\mathfrak{e}_2$ on avaldised. Tüübikorrektsuseks on vaja, et $\mathfrak{b}$ tüüp oleks `Bool` ning $\mathfrak{e}_1$ ja $\mathfrak{e}_2$ tüübid oleksid võrdsed.

Kui $\mathfrak{b}$ väärtus on True, siis avaldise (25) väärtus võrdub $\mathfrak{e}_1$ väärtusega; kui $\mathfrak{b}$ väärtus on False, siis avaldise (25) väärtus võrdub $\mathfrak{e}_2$ väärtusega; kui $\mathfrak{b}$ väärtus on $\perp$, siis avaldise (25) väärtus on $\perp$.

Näiteks deklaratsioon

```
tõeväärtusArvuks
  = \ x -> if x then 1 else 0
```
(26)

defineerib muutuja `tõeväärtusArvuks` väärtuseks funktsiooni, mis teisendab tõeväärtusi arvulisele kujule: tõene annab 1 ja väär 0.

Deklaratsioon

```
gm2
  = \ x y
      -> if x > 0 && y > 0
           then sqrt (x * y)
           else error "gm2: mittepos. argument"
```
(27)

defineerib muutuja `gm2` väärtuseks karritatud funktsiooni, mis võtab kaks arvulist argumenti ja kui mõlemad on positiivsed, annab välja nende geomeetrilise keskmise. Mittepositiivse argumendi korral katkeb arvutus veateatega. Siin muutuja `sqrt` tuleb moodulist Prelude ja tema väärtus on ruutjuure leidmise funktsioon.

1.1. Väärtustamine. Väärtustamaks avaldist (25), väärtustatakse kõigepealt avaldis $\mathfrak{b}$, kuni tuleb välja `True` või `False`. Vastavalt sellele jätkatakse kas $\mathfrak{e}_1$ või $\mathfrak{e}_2$ väärtustamisega.

2. Võrdlus teiste keeltega. Tingimusavaldis on lihtne ja sarnaneb teiste keelte *if*-konstruktsiooniga. See sarnasus on siiski mõnevõrra petlik.

Kõigepealt tuleb tähele panna, et Haskellis peab tingimusavaldisel alati olema nii *then*- kui ka *else*-haru. Teiseks pangem tähele, et tegemist on tõepoolest avaldisega, mitte mingi deklaratsiooni ega käsuga. Haskelli tingimusavaldise analoogiks C-s ja Javas on pigem konstruktsioon *<tingimus>* `?` *<avaldis>* `:` *<avaldis>* kui *if*-lause.

Et konstruktsioon kujul (25) on avaldis, on tema kasutamine niisama vaba kui avaldise kasutamine üldse. Näiteks võib ta asuda teise avaldise sees.

Defineerimaks muutuja `valiKääne` väärtuseks funktsiooni, mis võtab argumentideks arvu ja kaks sõnet ning annab välja esimese sõne, kui arv võrdub 1-ga, ja teise sõne ülejäänud juhtudel, kõlbab kood

```
valiKääne
  = \ n s1 sm                                          .
      -> show n ++ ' ' : if n == 1 then s1 else sm
```

Nüüd näiteks avaldise `valiKääne 1 "ärtu" "ärtut"` väärtus on "1 ärtu", avaldise `valiKääne 7 "ärtu" "ärtut"` väärtus aga "7 ärtut".

Pangem tähele, et tingimusavaldist ei pidanud alustama kohe noole järelt, mis oleks kaasa toonud mõlemale harule ühise koodi kordamise, vaid hargnemine on tõepoolest ainult seal, kus teda otseselt vaja.

3.2.2 Valikuavaldis

1. Hargnemine näidiste sobitamisega. Valikuavaldise puhul tehakse valik harude vahel, sobitades erinevaid näidiseid ühe kindla avaldise vastu. Valikuavaldis koosneb päisest kujul **`case`** $\mathfrak{e}$ **`of`**, kus $\mathfrak{e}$ on avaldis, mille vastu näidiseid sobitatakse, ja juhtude loendist. Iga juht koosneb näidisest ja vastavast paremast poolest, mis tuleb valida selle näidise sobitumise korral.

Lihtsaimal juhul koosneb parem pool noolemärgist **`->`** ja ühest avaldisest. Siis näeb valikuavaldis välja kujul

```
case a of
  p1 -> e1
  .........                                            (28)
  pl -> el
```

kus $\mathfrak{a}$ ja $\mathfrak{e}_1, \ldots, \mathfrak{e}_l$ on avaldised, $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ on näidised ning $l > 0$. Igas näidises $\mathfrak{p}_i$ esinevad muutujad on lokaalsed, nähtavad just i-nda juhu piires.

Tüübikorrektsuseks on vaja, et $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ oleksid kõik sama tüüpi nagu $\mathfrak{a}$ ja $\mathfrak{e}_1, \ldots, \mathfrak{e}_l$ oleksid omavahel ühesugust tüüpi.

Kui $\mathfrak{a}$ väärtus on normaalne (st pole $\perp$) ja i on vähim selline arv, mille korral $\mathfrak{p}_i$ sobitub $\mathfrak{a}$ väärtusega, siis avaldise (28) väärtuseks on $\mathfrak{e}_i$ väärtus olukorras, kus näidise $\mathfrak{p}_i$ väärtus võrdub avaldise $\mathfrak{a}$ väärtusega ja iga muutuja, mis esineb nii näidises $\mathfrak{p}_i$ kui ka avaldises $\mathfrak{e}_i$, on neis sama väärtusega. Samamoodi on asi, kui $\mathfrak{a}$ väärtus on $\perp$ ja näidis $\mathfrak{p}_1$ sobitub sellega. Kui üksiki näidistest $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ ei sobitu avaldise $\mathfrak{a}$ väärtusega, või $\mathfrak{a}$ väärtus on $\perp$ ja $\mathfrak{p}_1$ ei sobitu $\perp$-ga, siis on avaldise (28) väärtus $\perp$.

Näiteks deklaratsioon

```
üksLõppu
  = \ xs
      -> case xs of
           z : zs                                    (29)
             -> zs ++ [z]

           _
             -> []
```

defineerib `üksLõppu` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see on mittetühi, siis annab välja listi, mis on argumendist saadud esimese elemendi ümbertõstmisel listi lõppu; tühjal listil aga annab tühja listi. Tõepoolest, kui `xs` väärtus on mittetühi list, siis sobitub sellega valikuavaldise esimene näidis `z : zs`. Muutuja `z` saab väärtuseks listi pea, muutuja `zs` listi saba ning kogu valikuavaldis on samaväärne esimese juhu parema poolega `zs ++ [z]`. Kui `xs` väärtus on tühi list, siis esimene näidis ei sobitu, kuid teine sobitub ja kogu valikuavaldis on samaväärne teise juhu parema poolega `[]`.

Deklaratsioon

```
kordaEsimest
  = \ xs
      -> case xs of
           z : _                                     (30)
             -> z : xs

           _
             -> []
```

defineerib `kordaEsimest` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see pole tühi, siis annab välja listi, mis on argumentlistist saadud tema esimese elemendi veelkordsel lisamisel listi algusse; tühja listi puhul annab välja tühja listi.

Deklaratsioon

```
primLoenda
  = \ xs
      -> case length xs of
           0
             -> "Null"
           1
             -> "Üks"
           _
             -> "Mitu"
```
(31)

defineerib `primLoenda` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see on lõplik, siis annab välja teksti, mis ütleb, kas listis oli elemente null, üks või mitu; kui list on lõpmatu, siis on väärtuseks $\perp$ (arvutus jääb lõpmatult tööle).

Ülesandeid

127. Defineerida muutuja `arvTõeväärtuseks` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab tulemuseks tõeväärtuse True, kui $n = 1$, ja tõeväärtuse False, kui $n = 0$. Kui $n \notin \{0, 1\}$, siis peab rakendamine lõppema eestikeelse veateatega.

128. Defineerida muutuja `esipaar` väärtuseks funktsioon, mis võtab argumendiks listi ning, kui selles on vähemalt 2 elementi, annab väärtuseks paari kahest esimesest elemendist, vastasel korral lõpetab sobiva eestikeelse veateatega. Arvutuse maht ei tohi kasvada listi pikkuse kasvades.

129. Defineerida muutuja `esiAbs` väärtuseks funktsioon, mis võtab argumendiks arvude listi: mittetühja listi korral annab väärtuseks listi, mille saab argumentlistist esimese elemendi asendamisel tema absoluutväärtusega; muul juhul tühja listi.

1.1. Väärtustamine. Valikuavaldise (28) väärtustamine käib järgnevalt. Kõigepealt sobitatakse näidis $\mathfrak{p}_1$ avaldise $\mathfrak{a}$ vastu. Õnnestumise korral seotakse $\mathfrak{p}_1$ sobituselemendid ja kirjutatakse avaldis (28) ümber avaldiseks $\mathfrak{e}_1$. Vastasel korral $\mathfrak{p}_1$ sobituselemente ei seota. Kui seejuures $\mathfrak{a}$ väärtus on normaalne, siis heidetakse esimene juht kõrvale ja jätkatakse järjest samamoodi ülejäänud juhtudega. Kui juhud saavad otsa, siis antakse täitmisaegne viga.

Illustreerime arvutusprotsessi deklaratsiooni (31) järgi. Olgu väärtustatav avaldis

```
primLoenda (take 1 "ABC").
```
(32)

Kõigepealt kirjutatakse `primLoenda` deklaratsioonist (31) ümber deklaratsiooni parema poolega, milleks on lambdaavaldis. Vastavalt lambdaavaldise väärtustusskeemile tuleb nüüd sobitada näidist `xs` argumendi `take 1 "ABC"` vastu. Kuna muutuja sobitub alati, siis siin ei tehta ühtki väärtustussammu, `xs` seotakse avaldisega `take 1 "ABC"` ja kogu avaldis kirjutatakse ümber lambdaavaldise paremaks pooleks, mis on valikuavaldis. Edasi võetakse valikuavaldise päisest `length xs` ja sobitatakse esimest näidist `0` tema vastu, mistarvis väärtustatakse avaldist seni, kuni selgub sobitumine. See ei selgu enne, kui `length xs` on väärtustatud lõpuni. Kuna `xs` on seotud avaldisega `take 1 "ABC"`, mille väärtus on sõne "A", siis `length xs` väärtustamine annab tulemuseks `1`. Näidise `0` sobitamine sellega ebaõnnestub, seega proovitakse edasi järgmist juhtu. Seal on näidis `1`, mis sobitatakse juba eelmise juhu juures saadud tulemusega `1`. See õnnestub ja kogu avaldis kirjutatakse ümber sõnekonstandiks `"Üks"`. See on ka lõppväärtus.

Mõeldes väärtustamise käigu peale, saab selgeks, et `primLoenda` defineerimine deklaratsiooniga (31) on tegelikult väga ebaõnnestunud, sest valikuavaldise esimese näidise sobitamiseks tuleb argumentlisti pikkus välja arvutada, see nõuab aga tööd listi pikkusega võrdelises mahus. Kui list on väga pikk, siis töötab `primLoenda` väga kaua, enne kui midagi mõistlikku välja annab. Veel hullem, list võib olla lõpmatu või osaline, mispuhul `primLoenda` rakendamisel jääb arvutus lõpmatusse tsüklisse või lõpetab töö veateatega, samas kui kahe esimese elemendi leidmise järel võiks vastuse "Mitu" välja anda.

Sama funktsionaalsuse peaks realiseerima hoopis koodiga

```
primLoenda
  = \ xs
      -> case xs of
           []
             -> "Null" .                                   (33)
           [_]
             -> "Üks"
           _
             -> "Mitu"
```

Väärtustades selle definitsiooni järgi, selgub esimese näidise sobitumine või mittesobitumine `xs` väärtusega juba siis, kui käes on `xs` väärtust moodustav konstruktor (kas `:` või `[]`). Niipea kui on näiteks selgunud väärtuse jagunemine peaks ja sabaks, on teada, et esimene näidis ei sobitu. Sarnaselt piisab teise näidise sobitumise või mittesobitumise kindlakstegemiseks listi saba moodustav konstruktor kätte saada. Seetõttu annab see kood mõistliku tulemuse alati, kui `xs` väärtus on lõplik või lõpmatu list või selline osaline list, kus on vähemalt kaks elementi.

Teine asi, mida avaldise (32) deklaratsiooni (31) järgi väärtustamise käik selgitab, on põhjus, miks bottomiga sobitatakse ainult valikuavaldise esimest näidist. Kui avaldises (28) on a väärtus $\perp$ seetõttu, et tema väärtustamine tekitab täitmisaegse

vea, siis oleks ju võimalik see viga unustada ja proovida ka järgmisi näidiseid, millest mõni võib sobituda. Samas kui `a` väärtus on $\perp$ seetõttu, et tema väärtustamine jääb tühja lõpmatusse tsüklisse (nii juhtub, kui `primLoenda` argumendi väärtus on lõpmatu list, sest siis jääb pikkuse arvutamine tühjalt tööle), siis protsess esimesest näidisest kaugemale lihtsalt ei jõua. Et veaga lõpetamine ja lõpmatu arvutus on teooria järgi üks bottom kõik, siis ilmutatud viidatavuse nõude tõttu tuleb järelikult ka täitmisaegse vea puhul teiste näidiste kontrollimisest loobuda.

Ülesandeid

130. Kirjeldada detailselt avaldise `üksLõppu "ABC"` väärtustusprotsessi, kus `üksLõppu` on defineeritud deklaratsiooniga (29).

131. Kirjeldada detailselt avaldise `kordaEsimest [3, 5, 15]` väärtustusprotsessi, kus `kordaEsimest` on defineeritud deklaratsiooniga (30).

1.2. Hargnemine võrdluse tulemuse järgi. Kolmeks hargnemine võrdlustulemuse põhjal on tüüpiline olukord, kus kasutatakse valikuavaldist. Vaja läheb operaatorit `compare`, mille väärtus on karritatud funktsioon, mis võtab järjest kaks argumenti samast järjestusega tüübist ja annab välja nende võrdlustulemuse tüüpi Ordering kuuluva andmena. Selle tüübi andmeid esindavad konstruktorid `LT`, `EQ`, `GT`.

Vaid tingimusavaldisega tehes tuleks tingimusiavaldisi üksteise sisse panna, kuid valikuavaldis võimaldab piirduda ühe hargnemisega. Ühtlasi saavutatakse kahe võrdluse asemele üks, mis on oluline efektiivsuse kaalutlustel.

Olgu meil näiteks tarvis muutuja `absMax` väärtuseks defineerida karritatud funktsioon, mis võtab argumentideks kaks täisarvu ja kui nad on absoluutväärtuselt erinevad, siis annab tulemuseks absoluutväärtuselt suurema. Vastasel korral peab arvutus lõpetama töö veateatega.

Siin on vaja eristada kolm juhtu, mis tulenevad argumentide absoluutväärtuste võrdlemisest. Nõutud muutuja saame defineerida deklaratsiooniga

```
absMax
  = \ x y
      -> case compare (abs x) (abs y) of
           GT
             -> x
           LT
             -> y
           _
             -> error "absMax: abs.väärtused võrdsed"
```

.

Ülesandeid

132. Kasutades tüüpi Ordering, defineerida muutuja `võrdle` väärtuseks karritatud funktsioon, mis võtab argumentideks täisarvud x, y ja annab välja sõne "Võrdsed", "Järjestikused" või "Kauged" vastavalt sellele, kas $x = y$, x ja y on järjestikused täisarvud (ükskõik kummas järjekorras) või ei kehti kumbki neist tingimustest.

2. Samaväärsused. On selge, et kõik tingimusavaldised on võimalik asendada valikuavaldisega: kehtib samaväärsus

```
if 𝔟                    case 𝔟 of
  then 𝔢1      ≡          True
  else 𝔢2                   -> 𝔢1  .
                          _
                            -> 𝔢2
```

Näiteks võiks definitsiooni (27) asendada deklaratsiooniga

```
gm2
  = \ x y
      -> case x > 0 && y > 0 of
           True
             -> sqrt (x * y)
           _
             -> error "gm2: mittepos. argument"
```

Huvitaval kombel on ka argumendile rakendatud lambdaavaldis asendatav valikuavaldisega, kus valida on vaid ühe variandi vahel. Nimelt kehtib samaväärsus

```
(\ 𝔭 -> 𝔢) 𝔞   ≡   case 𝔞 of
                      𝔭 -> 𝔢   .
```

Osa programmeerijaid peab valikuavaldisega varianti kergemini loetavaks.

Ülesandeid

133. Kirjutada deklaratsioon (26) samaväärselt ümber valikuavaldisega.

134. Kirjutada avaldis (20) samaväärselt ümber valikuavaldisena.

3.2.3 Deklaratsioonisüsteemid

1. Süntaks ja tähendus.

1.1. Funktsionaalne vasak pool. Haskellis on olemas deklaratsioonid, mis erinevad seni nähtuist selle poolest, et vasak pool ei moodusta kokku üht näidist, vaid koosneb mitmest eraldi näidisest. Need näidised on kirjutatud nii nagu operaatori rakendamisel, kusjuures üks näidis on operaatori ja ülejäänud tema järjestikuste argumentide rollis. Sellist nimetatakse funktsionaalseks vasakuks pooleks.

Näidis, mis funktsionaalses vasakus pooles on operaatori rollis, saab olla ainult muutuja. Argumente märkivate näidiste reas ei tohi ükski muutuja korduda (sarnaselt mitmeargumendilise lambdaavaldise süntaksiga).

1.2. Üksik deklaratsioon funktsionaalse vasaku poolega. Deklaratsioon funktsionaalse vasaku poolega on niisiis kujul

$$\mathfrak{f}\ \mathfrak{p}_1\ \ldots\ \mathfrak{p}_l\ \texttt{=}\ \mathfrak{e}, \tag{34}$$

kus $\mathfrak{f}$ on prefikskujul muutuja ja $l > 0$, või kujul

$$(\mathfrak{p}_1\ \oplus\ \mathfrak{p}_2)\ \mathfrak{p}_3\ \ldots\ \mathfrak{p}_l, \tag{35}$$

kus $\oplus$ on infikskujul muutuja ja $l \geqslant 2$.

Deklaratsiooniga (34) või (35) defineeritakse seda muutujat, mis vasakus pooles on näidistele rakendatud (vastavalt $\mathfrak{f}$ ja $\oplus$). Tema väärtuseks saab teatav funktsioon, ülejäänud näidised aga märgivad tema argumente nende võtmise järjekorras. Argumendinäidiste muutujad on nähtavad vaid selles deklaratsioonis.

Infiksse rakendamise võib vastavalt tuntud reeglitele ümber kirjutada tavapärase prefiksse rakendamisega. Üksik deklaratsioon kujul (34) on samaväärne deklaratsiooniga

$$\mathfrak{f}\ \texttt{=}\ \texttt{\textbackslash}\ \mathfrak{p}_1\ \ldots\ \mathfrak{p}_l\ \texttt{->}\ \mathfrak{e},$$

kus vasak pool pole funktsionaalne.

1.3. Definitsioonid mitme deklaratsiooniga. Funktsionaalse vasaku poolega deklaratsioone, mis defineerivad sama operaatorit, võib olla mitu. Sellised deklaratsioonid moodustavad kokku deklaratsioonisüsteemi ja definee-

ritava muutuja väärtuse määrab süsteem koos. Kui rakendused on prefiksed, siis deklaratsioonisüsteemi on kujul

$$\begin{array}{l} \mathfrak{f}\ \mathfrak{p}_{1,1}\ \ \ldots\ \ \mathfrak{p}_{1,l} \\ \quad = \mathfrak{e}_1 \\ \ldots\ldots\ldots\ldots\ldots \\ \mathfrak{f}\ \mathfrak{p}_{k,1}\ \ \ldots\ \ \mathfrak{p}_{k,l} \\ \quad = \mathfrak{e}_k \end{array} \tag{36}$$

Sama süsteemi kuuluvad deklaratsioonid tunneb ära sellest, et näidis, mis teistele rakendub, on sama muutuja. Ülejäänud näidised ei pea olema deklaratsioonides samasugused. Küll aga peavad sama süsteemi deklaratsioonide vasakud pooled sisaldama ühepalju näidiseid. Süsteemi moodustavad deklaratsioonid peavad asetsema koodis järjest.

Tüübikorrektsuseks peavad erinevate deklaratsioonide vastavad näidised olema sama tüüpi ja kõik paremad pooled samuti.

Deklaratsioonisüsteemi mõte on juhtude läbivaatus, iga deklaratsioon märgib üht juhtu. Deklaratsioone loetakse järjekorras ülalt alla, argumendinäidiseid sobitatakse vasakult paremale lõpuni või esimese ebaõnnestumiseni. Kui argumentide väärtustamisel (mida tehakse niivõrd, kuivõrd näidiste sobitamine seda nõuab) ei teki täitmisaegseid vigu, siis esimene deklaratsioon, mille argumendinäidiste sobitamine õnnestub, läheb käiku kooskõlas ülaltoodud samaväärse ümberkirjutusega lambda kaudu. Kui sellist deklaratsiooni ei ole, antakse täitmisaegne viga.

Niiviisi annab deklaratsioonisüsteem (36) muutuja $\mathfrak{f}$ väärtuseks karritatud funktsiooni, mis võtab järjest argumendiks andmed $x_1, \ldots, x_l$. Kui i on vähim arv, mille korral i-nda deklaratsiooni kõik näidised sobituvad vastavate argumentidega ja ükski argument pole $\bot$, siis annab funktsioon tulemuseks $\mathfrak{e}_i$ väärtuse olukorras, kus näidistes $\mathfrak{p}_{i,1}, \ldots, \mathfrak{p}_{i,l}$ esinevad muutujad on seal ja avaldises $\mathfrak{e}_i$ sama väärtusega. Samamoodi on asi, kui mõni argument on $\bot$, kuid esimeses i deklaratsioonis kas selle argumendi näidis sobitub temaga või leidub oma argumendiga mittesobituv näidis enne selle argumendi näidist. Ülejäänud juhtudel on funktsiooni väärtuseks $\bot$.

Ülesandeid

135. Miks ei saa deklaratsioonisüsteemiga kirjeldatud muutuja väärtust esitada väärtuste kaudu, mille see muutuja omandaks selle süsteemi üksikutest deklaratsioonidest eraldi vastavalt ümberkirjutusele lambdaga?

2. Praktiline kasutus.

2.1. Võit lühiduses-ülevaatlikkuses. Funktsionaalse vasaku poolega deklaratsioonidega saab pisut lühemalt ja loetavamalt ümber kirjutada kõik varem käsitletud konstruktsioonide abil kirjutatud deklaratsioonid, mis defineerivad muutujate väärtuseks funktsioone.

Näiteks ruutfunktsioon sai muutuja `sqr` väärtuseks defineeritud deklaratsiooniga (21); uues süntaksis teeb sama deklaratsioon

```
sqr x
  = x * x.
```
(37)

Deklaratsioonid (22) ja (23) on samaväärselt ümber kirjutatavad vastavalt deklaratsioonideks

```
am2 x y
  = (x + y) / 2'

am3 x y z
  = (x + y + z) / 3.
```
(38)

või ka deklaratsioonideks

```
x `am2` y
  = (x + y) / 2'

(x `am3` y) z
  = (x + y + z) / 3.
```

Prefiksne või infiksne esinemine definitsioonis ei kohusta defineeritavat muutujat mujal samamoodi kasutama.

Definitsiooni, mille paremas pooles on valikuavaldis, mille päises olev avaldis langeb kokku mõne argumendinäidisega, kusjuures argumendinäidise muutujad mujal samas tähenduses ei esine, saab mehhaaniliselt deklaratsioonisüsteemiks ümber kirjutada.

Näiteks deklaratsiooniga (29) defineeritud muutuja `üksLõppu` defineerib samaväärselt süsteem

```
üksLõppu (z : zs)
  = zs ++ [z]
üksLõppu _
  = []
```

Ülesandeid

136. Otsida Hugsi teegist üles muutujate `fst`, `snd`, `head`, `tail`, `const` definitsioonid ja saada neist aru.

137. Defineerida uuel viisil muutuja `erinevus` väärtuseks ülesandes 116 kirjeldatud funktsioon.

138. Võimalikult lühikese ja võimalikult vähe muutujaid kasutava deklaratsiooniga kujul (34) defineerida muutuja `imelik` väärtuseks ülesandes 119 kirjeldatud funktsioon.

139. Defineerida operaator `takeLõpust`, mis töötab nagu `take`, aga elemente võetakse listi lõpust. Näiteks `takeLõpust 2 [3, 2, 4]` väärtustamine peab andma `[2, 4]`.

140. Defineerida muutuja `karritErinevus` väärtuseks ülesandes 116 kirjeldatud funktsiooni karritamise tulemus. kasutades defineeritavat muutujat prefiksselt.

141. Modifitseerida ülesandes 140 kirjutatud definitsiooni nii, et defineeritav muutuja oleks kasutatud infiksselt.

142. Muuta ülesandes 140 defineeritud muutuja infikskuju prioriteet ja assotsiatiivsus sarnaseks plussi ja miinusega.

143. Lahendada ülesanded 127, 128, 129 deklaratsioonisüsteemiga.

144. Kirjutada muutuja `primLoenda` definitsioon (33) samaväärselt ümber deklaratsioonisüsteemiga.

145. Kirjutada muutuja `kordaEsimest` definitsioon (30) samaväärselt ümber mitme deklaratsiooniga, kasutades samu muutujaid samas tähenduses.

2.2. Hargnemine mitme näidise järgi. Deklaratsioonisüsteemi eelised valikuavaldise ees ilmnevad eriti juhul, kui hargnemisotsuseid tehes tuleb arvestada korraga mitme argumendi väärtusi.

Näiteks süsteem

```
risti (x : _) (_ : ys)
  = x : ys
risti _       _
  = []
```
(39)

defineerib muutuja `risti` väärtuseks karritatud funktsiooni, mis võtab kaks argumentlisti: kui mõlemad on mittetühjad, siis annab välja listi, mille pea tuleb esimesest ja saba teisest listist; vastasel korral annab välja tühja listi. Esimene deklaratsioon läheb käiku parajasti juhul, kui mõlemad argumendid on mittetühjad listid, sest parajasti siis sobituvad mõlemad argumendinäidised. Ülejäänud juhtudel läheb käiku teine deklaratsioon, sest seal argumentidele tingimusi ei seata.

Et valikuavaldisega sama saavutada, tuleks argumendid üheks järjendiks kokku võtta, sest valikuavaldise päisesse saab kirjutada ainult ühe avaldise. See aga viib liigsete andmestruktuuride moodustamiseni ka arvutuse käigus ning kood jookseks mõnevõrra aeglasemalt.

Ülesandeid

146. Defineerida muutuja `pead` väärtuseks karritatud funktsioon, mis võtab argumentideks kaks listi ja annab väärtuseks listi, mille elementideks on ilma midagi kordamata kõik need andmed, mis esinevad esimese või teise listi esimese elemendina.

147. Kirjutada oma Haskell-faili muutuja `risti` definitsioon (39) ja veenduda, et see töötab. Seejärel asendada teine deklaratsioon deklaratsiooniga

```
risti
  = \ _ _ -> []
```

Tutvuda veateatega, mis tekib.

148. Kas saame süsteemiga (39) samaväärse definitsiooni, kui viime korraga mõlema deklaratsiooni argumendid lambdaavaldisega paremasse poolde (nii nagu ülesandes 147 tehti teise deklaratsiooniga)?

3. Funktsionaalse vasaku poolega deklaratsioonid tüübitasemel. Kuna tüüpidel on võimalikud ainult muutujanäidised (Haskelli laiendused aktsepteerivad ka jokkerit), ei ole tüübisünonüümide definitsioonides hargnemine võimalik. Siiski on võimalik kasutada funktsionaalset vasakut poolt.

Näiteks deklaratsiooniga

```
type Maatriks a
  = [[a]]
```

võime defineerida muutuja `Maatriks` väärtuseks tüübifunktsiooni, mis igal argumendil A annab väärtuseks A-tüüpi väärtustega listide listide tüübi. (Mõtteks võib olla käsitleda listide elementliste maatriksi ridadena.)

3.2.4 Valvurikonstruktsioon

Deklaratsiooni või valikuavaldise juhu paremaks pooleks võib lisaks taolistele, mida seni oleme kasutanud, olla **valvurikonstruktsioon**[1].

1. Üldpõhimõte. Deklaratsiooni parema poolena näeb valvurikonstruktsioon välja kujul

$$
\begin{array}{l}
\quad \texttt{|}\ g_1\ \texttt{=}\ e_1 \\
\ldots\ldots\ldots\ldots \\
\quad \texttt{|}\ g_l\ \texttt{=}\ e_l
\end{array}
$$

Selle konstruktsiooni mõte on esitada üleminek sõltuvalt lisatingimustest. Vasakul olevaid süntaktilisi üksusi g_i nimetatakse **valvuriteks**; valvur kujutab endast tõeväärtusetüüpi avaldist, tingimust. Igale valvurile vastavas lokaalses paremas pooles on avaldis, mis tuleb valida juhul, kui see valvur on esimene, mis väärtustub tõeseks, ja eelmiste väärtused on normaalsed. Mõistagi peavad kõik lokaalsed paremad pooled olema ühesugust tüüpi.

Valikuavaldise juhu parema poolena näeb valvurikonstruktsioon välja samamoodi, ainult võrdusmärkide asemel on noolemärgid `->`. Ka tähendus on täpselt sama nagu deklaratsiooni parema poolena.

Näitena valvuritega deklaratsioonist defineerime muutuja `arvuKlass` väärtuseks funktsiooni, mis arvulisel argumendil annab väärtuseks teksti “Negatiivne”, “Nullilähedane” ja “Suur”, kui argument on vastavalt negatiivne, mittenegatiivne 1-st väiksem, või 1 või suurem. Deklaratsiooniks sobib

```
arvuKlass x
  | x < 0
    = "Negatiivne"
  | x >= 0 && x < 1   .                    (40)
    = "Nullilähedane"
  | x >= 1
    = "Suur"
```

Arvutus jõuab konkreetse valvurini ainult siis, kui ükski eelnev valvur samas komplektis pole sobinud. Seega deklaratsiooni (40) teine valvur `x >= 0 && x < 1` väärtustatakse ainult siis, kui esimene valvur on juba väärtustatud ja on selgunud, et `x` väärtus negatiivne ei ole. Järelikult poleks teises valvuris enam mõtet uurida tingimust `x >= 0`, kuna selle väärtus on kindlasti tõene.

[1] Siin kirjeldatakse valvurikonstruktsioon vastavalt standardile Haskell 98. Uues standardis Haskell 2010 on seda konstruktsiooni üldistatud nn **näidisvalvurite** sissetoomisega.

Analoogiliselt tuleb kolmas valvur `x >= 1` kontrollimisele ainult siis, kui esimese kahe valvuri kohta on juba selgunud, et nende väärtus on väär. See aga tähendab, et `x` väärtus pole ei negatiivne ega 1-st väiksem mittenegatiivne, ehk ta on 1-st suurem või sellega võrdne. See on aga just see tingimus, mida kolmas valvur kontrollib. Järelikult poleks kolmanda valvurini jõudes vaja enam midagi kontrollida.

Neist kaalutlustest lähtuvalt saame deklaratsiooni (40) lihtsustada, võites ühtaegu koodi lühiduses kui ka arvutuse kiiruses. Viimase valvuri asemele võiksime kirjutada mistahes avaldise, mille väärtus on True, näiteks `True` enda, et arvutuste hulka võimalikult vähendada; loetavuse huvides kasutatakse sellises olukorras moodulis Prelude defineeritud muutujat `otherwise` väärtusega True. Kokkuvõttes saame deklaratsiooni

```
arvuKlass x
  | x < 0
    = "Negatiivne"
  | x < 1
    = "Nullilähedane"
  | otherwise
    = "Suur"
```

Selle järgi arvutades tuleb teha maksimaalselt 2 võrdlust.

Märgime, et muutuja `arvuKlass` oleks siiski võimalik elegantselt samaväärselt defineerida ka valikuavaldise abil, kasutades võrdlemist muutuja `compare` abil. Märkame, et kolm juhtu vastavad sellele, kas argumendi täisosa on negatiivne, null või positiivne. Täisosafunktsioon on mooduli Prelude muutuja `floor` väärtuseks. Seega saame deklaratsiooni

```
arvuKlass x
  = case compare (floor x) 0 of
      LT
        -> "Negatiivne"
      EQ
        -> "Nullilähedane"
      _
        -> "Suur"
```

2. Tagasivõtmine. Ilmselt on valvurikonstruktsioonil võrreldes teiste hargnemiskonstruktsioonidega mugavuseelis, kui tingimusi on rohkem. Kuid valvurikonstruktsiooniga tuleb kaasa ka üks sisuline omapära, tagasivõtmine.

Nimelt kui arvutuse käigus väärtustuvad valvurikonstruktsiooni kõik valvurid vääraks, siis loetakse vastav deklaratsioon või valikuavaldise juht mittesobivaks ja valitakse järgmine, justkui selle deklaratsiooni või juhu näidiste

sobitamine oleks ebaõnnestunud. Niisiis, erinevalt senistest hargnemistest, ei anna valvurikonstruktsiooni kõigi harude mittesobivus automaatselt täitmisaegset viga, vaid hargnemiste puus ühe taseme võrra tagasi pöördumise.

Vaatleme valvurikonstruktsiooni sisaldavat definitsiooni

```
listiKlass (z : _)
  | z > 0
    = ":)"
  | z == 0
    = ":|"
listiKlass _
  = ":("
```

.

Muutuja `listiKlass` rakendamine mingile listitüüpi avaldisele ⌊ käib järgnevalt. Kui ⌊ väärtus on mittetühi list, siis sobitub temaga esimese deklaratsiooni argumendinäidis ja `z` saab väärtuseks listi pea. Edasi sõltub asi sellest, milline see listi pea on. Kui ta on positiivne, siis esimene valvur väärtustub tõeseks ja rakendamise tulemus on sõne ":)". Kui listi pea on null, siis esimene valvur väärtustub vääraks, teine aga tõeseks, mistõttu rakendamise tulemus on sõne ":|". Kui listi pea on negatiivne, siis tõeseks väärtustuvat valvurit ei leidu, kogu deklaratsioon ja muutuja `z` sidumine hüljatakse ja võetakse järgmine deklaratsioon jokkeriga. Seal takistusi ei teki ja rakenduse tulemus on ":(".

Kasutades laiska näidist, saab sama funktsiooni kirjeldada siiski ka ühel tasemel hargnemisega. Sobib deklaratsioon

```
listiKlass xs@ ~(z : _)
  | null xs || z < 0
    = ":("
  | z > 0
    = ":)"
  | otherwise
    = ":|"
```

. (41)

Vaatame, kuidas toimub `listiKlass` ⌊ väärtustamine definitsiooni (41) korral. Alguses seotakse näidised `xs` ja `z : _` avaldisega ⌊, siis minnakse valvurikonstruktsiooni juurde. Kui ⌊ väärtus on tühi, väärtustub esimese valvuri disjunktsiooni vasak pool ja seega ka kogu valvur tõeseks ilma disjunktsiooni teist poolt uurimata ning `listiKlass` ⌊ väärtuseks saadakse ":(". Tänu laisale väärtustamisele niisugune definitsioon töötab: kui oleks nõutud ka `||` parema argumendi väärtustamine, siis järgneks siin viga, kuna muutujat `z` pole võimalik siduda. Kui ⌊ väärtus on mittetühi, siis esimese valvuri disjunktsiooni vasak pool väärtustub vääraks ja seega asutakse väärtustama disjunktsiooni paremat poolt. See nõuab muutuja `z` väärtust,

seepärast sobitatakse näidis `z : _` temaga seotud avaldise vastu, milleks on `ɭ`. Kuna `ɭ` väärtus on mittetühi, siis see õnnestub ja `z` seotakse avaldisega, mille väärtus on listi pea. Edasi on juba selge — käitutakse vastavalt `z` väärtusele.

Ülesandeid

149. Defineerida muutuja `esiNegSgn` väärtuseks funktsioon, mis võtab argumendiks arvude listi: kui selle esimene element on negatiivne, siis annab väärtuseks listi, mille saab argumentlistist esimese elemendi asendamisel arvuga -1, kõigil ülejäänud juhtudel annab välja argumentlisti enda.

150. Lahendada ülesanne 146 valvurikonstruktsiooni abil.

151. Kas deklaratsioonis (41) disjunktsiooni poolte vahetamisel saadakse samaväärne deklaratsioon?

3.3 Suuremad konstruktsioonid

Selles jaotises käime läbi keerulisemad süntaktilised konstruktsioonid, mis võivad endas sisaldada mitmeid deklaratsioone.

3.3.1 Lokaalsete deklaratsioonidega konstruktsioonid

Lokaalsed deklaratsioonid võimaldavad võtta muutujaid kasutusele lokaalses tähenduses.

Põhiline olukord, kus lokaalne deklaratsioon on abiks, on ühe ja sama mittetriviaalse avaldise korduv esinemine samas deklaratsioonis. Sama avaldise kordumisel koodis võidakse seda avaldist arvutuse käigus korduvalt väärtustada; kui aga siduda see avaldis muutujaga ja asendada selle avaldise esinemised selle muutujaga, siis see väärtustamine toimub ülimalt üks kord. Isegi kui avaldise korduva väärtustamise ohtu pole, näiteks esineb ta hargnemiskonstruktsiooni eri harudes, on suurema avaldise korduv esinemine klassikaline koodikordus ja sellisena taunitav.

Lokaalsete muutujate sissetoomine võib aidata ka lihtsalt koodi loetavust parandada. Kui on vaja kirjutada mõni väga pikk ja keeruline avaldis, siis võib selguse mõttes tähtsamad alamavaldised välja tuua ja lokaalsete muutujatega siduda (soovitavalt valides muutujatele vastavate vaheetappide tulemust sisuliselt iseloomustavad nimed).

1. *Let*-avaldis. Põhilisim konstruktsioon lokaalsete deklaratsioonide kirjutamiseks on *let*-avaldis kujul

```
let
  𝔡₁
  ..
  𝔡ₗ .
in
𝔢
```

Tema väärtuseks on avaldise $\mathfrak{e}$ väärtus tingimusel, et deklaratsioonides $\mathfrak{d}_1, \ldots, \mathfrak{d}_l$ defineeritud muutujate väärtused saadakse neist deklaratsioonidest. Deklaratsioonides $\mathfrak{d}_1, \ldots, \mathfrak{d}_l$ defineeritud muutujad on selles tähenduses nähtavad parajasti kogu selles *let*-avaldises.

Näiteks deklaratsioonis

```
letNäide x
  = let
      y = x + 0.5
      kuup x
        = x * x * x
      a = kuup x + 3 * x * y + 2 * kuup y
      b = sin x - 3 * sin x * cos y + cos x
    in
    b / a * 100
```

on muutujaga `y` seotud korduvalt vajaminev avaldis `x + 0.5`, et vältida sama arvutuse ja koodi kordumist. Kahes kohas kasutatava kuupitõstmise jaoks on tehtud lokaalne operaator `kuup`. Samuti tuuakse eraldi muutujatega `a` ja `b` välja arvutatava avaldise pikemad alamavaldised. Paneme tähele, et operaatori `kuup` definitsioonis esineb muutuja `x` lokaalses, välise operaatori `letNäide` argumendist erinevas tähenduses.

Ülesandeid

152. Defineerida muutuja `summanali` väärtuseks funktsioon, mis võtab argumendiks arvupaari ja annab väärtuseks selle paari komponentide summa siinuse ja summa enda suhte. Sama summat ei tohi arvutada korduvalt. Kõik vajalikud oma abimuutujad defineerida lokaalselt.

153. Defineerida muutuja `pooleks` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab välja paari kahest täisarvust, mille summa on n ja vahe on 0 või 1.

154. Defineerida muutuja `õigeProperFraction` väärtuseks funktsioon, mis võtab argumendiks reaalmurrulist tüüpi (st klassi RealFrac kuuluvat tüüpi) arvu x ja annab tulemuseks paari x täisosa ja murdosaga, kus murdosa on alati mittenegatiivne.

2. *Where*-konstruktsioon. Kui mõnd avaldist kasutatakse korduvalt mitmes valvuris või mitmele valvurile vastavates lokaalsetes paremates pooltes, siis pole *let*-avaldisega võimalik olukorda parandada, kuna valvurikonstruktsioon ei moodusta avaldist.

Selleks puhuks on Haskellis olemas *where*-konstruktsioon, mis võimaldab defineerida lokaalseid muutujaid, mis on nähtavad üle kogu parema poole, koosnegu ta siis ühest avaldisest või valvurikonstruktsioonist. *Where*-konstruktsioon kujutab endast deklaratsiooni või valikuavaldise juhu parema poole jätku, mis kirjutatakse kogu parema poole lõppu, kujul

```
where
  d1
  ..   .
  dl
```

Näide tüüpilisest olukorrast, kus *where*-konstruktsioon on kasulik, on deklaratsioon

```
veerand x
  | a > 0 && b > 0
    = "I" ++ v
  | a > 0 && b < 0
    = "II" ++ v
  | a < 0 && b < 0
    = "III" ++ v
  | a < 0 && b > 0    .
    = "IV" ++ v
  | otherwise
    = "vahepeal"
  where
    a = sin x
    b = cos x
    v = " veerandis"
```

Muutuja `veerand` väärtuseks saab funktsioon, mis võtab argumendiks ujukomaarvu ja, interpreteerides seda nurga suurusena, tuvastab, millises veerandis on vastav nurk. Lokaalsed muutujad `a` ja `b` leiavad kasutust mitmes valvuris, `v` aga erinevates lokaalsetes paremates pooltes.

Ülesandeid

155. Defineerida muutuja `sümbolid` väärtuseks funktsioon, mis võtab argumendiks täisarvu ja annab väärtuseks teksti, mis ütleb, kas sellele arvule kooditabelis vastav sümbol on suurtäht, väiketäht, number või midagi muud; kui aga arv pole lõigus 0-st 255-ni, siis anda hoopis seda ütlev tekst. Ühtki avaldist ei tohi arvutada korduvalt. Kõik vajaminevad oma abimuutujad defineerida lokaalselt. Vajalikke kontrollfunktsioone otsida moodulist Data . Char.

156. Defineerida muutuja `nihe` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja sümboli c: kui c on ladina tähestiku täht, siis annab väärtuseks c-st ladina tähestikus n kohta tagapool oleva tähe; vastasel korral annab väärtuseks c. Tähestikku vaatame tsüklilisena, st z järel tuleb jälle a. Suurtähe puhul peab välja tulema suurtäht, väiketähe puhul väiketäht. Püüda koodikordust vältida.

157. Lahendada ülesanne 154 valvurikonstruktsiooni abil.

3. Arvutuse käik. Nii *let*-avaldist kui ka *where*-konstruktsiooni iseloomustab asjaolu, et arvutus ei puutu lokaalseid deklaratsioone enne, kui mõnd neis defineeritud muutujat vaja läheb. Teisi sõnu, lokaalsete deklaratsioonidega käitutakse nagu globaalsetegagi.

See tähendab muuhulgas, et

```
let
  p = a
in
b
```
(42)

ei ole alati samaväärne avaldisega

```
(\ p -> b) a,
```
(43)

ehkki paljudel juhtudel on. Vahe tuleb näidise sobitamise ajastusest, nagu lambdaavaldise väärtustamise juures sai juba selgitatud. Praktiline erinevus puudub juhul, kui näidis `p` on selline, mille sobitamine tähendabki ainult tema sidumist avaldisega: muutuja, jokker või laisk näidis.

Erinevus võib avalduda ka avaldise väärtuses. Kui näiteks `p = x : xs`, `a = []` ja `b = 5`, siis avaldise (42) väärtus on 5, kuid (43) väärtus on $\bot$, tema väärtustamine lõpeb näidise sobitamise ebaõnnestumise tõttu veateatega.

3.3.2 Listikomprehensioon

Viimased süntaktilised konstruktsioonid avaldiste moodustamiseks, mida vaatame, kannavad **komprehensioonsüntaksi** nime. Selle süntaksiga saab elegantse kompaktsusega väljendada küllaltki keerulisi andmeid.

Haskellis on kaks komprehensioonsüntaksit — listikomprehensioon ja monaadikomprehensioon[1]. Vaatleme kõigepealt esimest.

1. Üldist. **Listikomprehensioon** on analoogne matemaatikas tuntud hulgakomprehensiooniga, millele lisandub arvutuslik aspekt. **Hulgakomprehensioon** on viis esitada hulki teatud kitsenduste kaudu nagu näiteks kujus $\{x^2 \mid 1 \leqslant x \leqslant 9, x \neq 5\}$, mis märgib hulka kõigi ühekohaliste 5-st erinevate positiivsete täisarvude ruutudest.

Haskelli listikomprehensioon on ka välimuselt sellega sarnane, olles kujul

$$\texttt{[}\mathfrak{a}\ \texttt{|}\ \mathfrak{q}_1\texttt{,}\ \ldots\texttt{,}\ \mathfrak{q}_l\texttt{]}, \tag{44}$$

kus $\mathfrak{a}$ on avaldis ning $l > 0$ ja iga $\mathfrak{q}_i$ on kas nn generaator, valvur või lokaalne deklaratsioon.

Generaator on süntaktiline üksus kujul

$$\mathfrak{p}\ \texttt{<-}\ \mathfrak{e},$$

kus $\mathfrak{p}$ on näidis ja $\mathfrak{e}$ on avaldis. Generaator defineerib näidises esinevad muutujad, kuid seejuures on iga generaatori näidise muutujad nähtavad vaid järgmistes generaatorites, mitte selles generaatoris ega eelmistes.

Avaldise (44) püstkriipsust vasakul olevas avaldises $\mathfrak{a}$ on nähtavad kõigi generaatorite defineeritud muutujad. Listikomprehensioonis peab iga generaatori parema poole avaldis olema listitüüpi, kuid elemenditüüp võib erinevates generaatorites olla erinev. Kogu avaldise (44) tüüp on listitüüp, kus elemenditüüp võrdub avaldise $\mathfrak{a}$ tüübiga.

2. Generaatoritega listikomprehensioon. Vaatame algul lihtsustatud, kuid põhilist olukorda, kus iga $\mathfrak{q}_i$ kujus (44) on generaator, st avaldis on kujul

$$\texttt{[}\mathfrak{a}\ \texttt{|}\ \mathfrak{p}_1\ \texttt{<-}\ \mathfrak{e}_1\texttt{,}\ \ldots\texttt{,}\ \mathfrak{p}_l\ \texttt{<-}\ \mathfrak{e}_l\texttt{]}. \tag{45}$$

[1] Uus standard Haskell 2010 toob sisse näidisvalvurid — valvurikonstruktsiooni üldistuse, mis kujutab endast omamoodi kolmandat komprehensioonsüntaksit.

Avaldise (45) väärtus on list kõigist sellistest väärtustest, mille avaldis $\mathfrak{a}$ omandab, kui näidised $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ omandavad kõikvõimalikel viisidel väärtused vastavate avaldistega $\mathfrak{e}_1, \ldots, \mathfrak{e}_l$ kirjeldatud listide elementide seast nii, et kõik näidised ka sobituvad oma väärtusega. Iga variant kõigi näidiste sobitumiseks vastavate listide mingite elemendiga annab tulemuslisti ühe elemendi. Erandiks on olukord, kus näidis ei sobitu seetõttu, et listi element on $\perp$, või kui mõni list ise on osaline; siis jääb arvutus sealkohal toppama ja tulemuslist jääb osaliseks.

Näidist sobitatakse listi elementidega nende seal esinemise järjestuses, vastavas järjestuses on ka tulemuslisti elemendid.

Näiteks avaldise

```
[x * x | x <- [0 .. ]]
```
(46)

väärtuseks on list kõigi naturaalarvude ruutudest kasvavas järjestuses. Tõepoolest, siin on ainult üks generaator, kus olev näidis `x` võrdsustatakse järjest paremas pooles kirjeldatud listi iga elemendiga ehk kõigi naturaalarvudega. Näidis alati sobitub, kuna ta on muutuja. Seega tulemuslisti tekib `x * x` väärtus `x` iga naturaalarvulise väärtuse jaoks ehk naturaalarvude ruudud.

Analoogselt on avaldise

```
[(x , x * x) | x <- [1 .. 9]]
```
(47)

väärtuseks list kõigist ühekohalistest arvudest paaris oma ruuduga.

Kui on vaja funktsiooni, mis võtab argumendiks listide listi ja annab tulemuseks nende peade listi, jättes tühjad listid vahele, siis komprehensioonsüntaks on sobiv valik, võimaldades elegantselt vältida ilmutatud hargnemisi listi tühjuse- mittetühjuse järgi. Deklaratsioon

```
pead xss
  = [x | x : _ <- xss]
```
(48)

defineerib just sellise funktsiooni muutuja `pead` väärtuseks. Tõepoolest, näidis `x : _` võrdsustatakse argumentlisti iga elemendiga; tühjade listidega näidis ei sobitu ja need jäävad vahele, mittetühjadega aga sobitub ja need annavad `x` väärtuseks oma pea. Igast sellisest juhtumist tekib `x` väärtus ehk seesama pea ka tulemuslisti.

Näiteks `pead ["ABC", "", "hi", "!", ""]` väärtus on “Ah!”.

Kui aga huvitavad mittetühjade elementide pikkused, siis sobib deklaratsioon

```
mittetühjadePikkused xss
  = [length xs | xs@ (_ : _) <- xss].
```
(49)

Näiteks on `mittetühjadePikkused ["ABC", "", "hi", "!", ""]` väärtus lõplik list elementidega 3, 2, 1.

Mitme generaatori puhul tekitatakse väärtuste komplektid grupeerituna esimese generaatori näidise väärtuste kaupa, gruppides omakorda teise generaatori näidise väärtuste kaupa jne. Mida kaugem generaator, seda kiiremini ta väntab. Vastavas järjestuses on ka tulemuslisti elemendid.

Mitme generaatoriga listikomprehensiooni näitena võtame avaldise

```
[(x , y) | x <- [1 .. 9], y <- [1 .. 5]],
```

mille väärtus on list täisarvupaaridest $(x\,, y)$, kus $1 \leqslant x \leqslant 9$ ja $1 \leqslant y \leqslant 5$. Tänu sellele, et mõlemas generaatoris on listi elemendid kasvavalt järjestatud, on ka tulemuses paarid järjestatud — kõigepealt vasaku ja seejärel parema komponendi järgi, sest vasak komponent tuleb esimesest ja parem teisest generaatorist.

Kuna eespoolsete generaatorite näidistes esinevad muutujad võivad tagapoolsete generaatorite avaldistes esineda, siis saab kirjutada näiteks avaldise

```
[(x , y) | x <- [1 .. 9], y <- [x .. 9]].
```
(50)

Selle väärtustamisel on `x` iga uue väärtuse korral uus ka list, kust `y` väärtusi omandab, sisaldades täisarvud `x` jooksvast väärtusest 9-ni. Seega on avaldise (50) väärtus list kõigist ühekohaliste arvude paaridest, kus vasak komponent ei ületa paremat.

Ülesandeid

158. Lahendada ülesanne 28 komprehensioonsüntaksi abil.

159. Kirjutada listikomprehensioonavaldis, mille väärtuseks on sõnede list, mille elemendid on kahekohalised kümnendarvud suuruse järjestuses.

160. Modifitseerida ülesande 159 lahendust nii, et kõigi kahekohaliste kümnendarvude asemel on kõik kahekohalised kuueteistkümnendarvud.

161. Kirjutada avaldis, mille väärtus on kahes suunas lõpmatu korrutustabel listide listina, st listi nr n element nr m peab olema $n \cdot m$, kus nummerdamine käib alates 0-st.

162. Kasutades ära ülesandes 156 defineeritud muutujat `nihe`, defineerida muutuja `nihkesiffer` väärtuseks karritatud funktsioon, mis võtab argumendiks täisarvu n ja sõne s ning annab väärtuseks sõne, mille saab sõnest s iga ladina tähe asendamisel temast tsüklilises ladina tähestikus n koha võrra tagapool oleva tähega.

163. Defineerida muutuja `leksCompare` väärtuseks karritatud funktsioon, mis võtab argumendiks kaks sõnet ja annab välja nende võrdlemise tulemuse tähestiku järjekorra alusel andmena tüübist Ordering. Sama tähe suur- ja väikevariandid lugeda ekvivalentseks, kuid kui selliselt võttes on tegemist sama sõnaga, siis lugeda suurtähed väiketähtedest eespool seisvaks.

164. Kui deklaratsioonis (48) muuta näidis `x : _` tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust?

165. Kui deklaratsioonis (49) muuta näidis `(_ : _)` tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust? Kas ja kuidas väärtus muutub, kui laisaks muuta terve generaatori vasak pool `xs@ (_ : _)`?

3. Listikomprehensioon valvurite ja lokaalsete deklaratsioonidega. Valvurid ja lokaalsed deklaratsioonid listikomprehensioonis ei muuda senitutvustatud reegleid muutujate tähendusulatuse ja väärtustamise kohta.

Valvur on, nagu ka valvurikonstruktsioonis, tõeväärtusetüüpi avaldis. Tema roll on esitada näidiste väärtuste komplektile lisakitsendusi: tulemuslist peab sisaldama elemente ainult selliste komplektide jaoks, mille puhul valvurid väärtustuvad tõeseks. Põhimõtteliselt on valvur `g` listikomprehensioonis samaväärne generaatoriga `True <- [g]`.

Lokaalsed deklaratsioonid võimaldavad sisse tuua uusi muutujaid ilma neid listist genereerimata. Lokaalsed deklaratsioonid algavad võtmesõnaga **`let`**.

Vaatame uuesti avaldist (47), mille väärtus on list kõigi ühekohalistest arvudest paaris oma ruuduga. Kui meid arvu 5 ruut ei huvita, siis võime selle listist välja jätta, lisades valvuri, mis sellisel juhul väärtustub vääraks. Saame avaldise

```
[(x , x * x) | x <- [1 .. 9], x /= 5].
```
(51)

Listikomprehensiooni kasutamisel tuleb arvestada, et arvutuskiirus võib oluliselt sõltuda generaatorite ja valvurite järjestusest. Valvurid, mis osa variante välistavad, on kasulik panna nii ette kui võimalik, sest väärtustamisel loetakse komprehensiooni vasakult paremale ja variante, kus valvur annab väära või näidis mõne elemendiga ei sobitu, ei hakatagi lõpuni koostama, vaid asutakse kohe järgmise elemendi kallale.

Avaldise (50) saab samaväärselt ümber kirjutada kujul

```
[(x , y) | x <- [1 .. 9], y <- [1 .. 9], x <= y].
```
(52)

Võrreldes aga avaldiste (50) ja (52) väärtustussammude arve, saab selgeks, et arvutuslikus mõttes pole nad sugugi võrdväärsed, (50) on efektiivsem. Tõepoolest: avaldise (52) puhul genereeritakse muutujate `x` ja `y` väärtustuseks kõik variandid ühekohaliste arvudega, millest valvur ligemale pooled kõlbmatuks tunnistab, samas kui avaldise (50) korral genereeritaksegi ainult need väärtustused, mis annavad tulemuslisti elemendi.

Samal põhjusel ei ole avaldis (47) üldse samaväärne avaldisega

```
[(x , x * x) | x <- [1 .. ], x <= 9].
```
(53)

Avaldise (53) väärtustamist pole võimalik lõpetada, sest generaator annab muutujale `x` muudkui uusi väärtusi. Tulemuslist on osaline, ehkki sisaldab parajasti kõik need elemendid mis avaldise (47) väärtus.

Viimase, pikema näitena olgu meil millegipärast vaja funktsiooni, mis võtab argumendiks täisarvude listi ja annab tulemuseks sõne, mis koosneb plokkidest, millest igaühe alguses on number 1 ja selle järel niimitu numbrit 0, nagu näitab argumentlisti järjekordne element, kusjuures negatiivseid ja liiga suuri (ütleme alates 8-st) elemente tuleb ignoreerida.

Nõutud plokid on lihtne leida kümne astmetena. Arvutuse käiku tundmata võime kirjutada korrektse, kuid ebaefektiivse definitsiooni

```
plokid ns
  = [c | n <- ns, c <- show (10 ^ n), 0 <= n && n < 8]
```
(54)

Selle korral genereeritakse näiteks avaldise `plokid [1, 99999, 4]` väärtustamise käigus muutujale `c` väärtused ka `n` väärtuse 99999 puhul, mida järgnev valvur ei luba. See tühi lisatöö kulutab meeletu aja, sest `c` saab järjest väärtuseks kõik numbrid 100000-kohalisest arvust ja iga kord väärtustatakse valvur uuesti.

Kuna see valvur muutujat `c` ei sisalda, siis saab ta tuua eelneva generaatori ette. Tekib definitsioon

```
plokid ns
  = [c | n <- ns, 0 <= n && n < 8, c <- show (10 ^ n)]
```
(55)

mille korral leitakse avaldise `plokid [1, 99999, 4]` väärtus välkkiirelt, sest numbreid genereerima asutakse ainult juhul, kui valvur on läbi lubanud.

Definitsioonil (54) on ka muid puudusi. Näiteks avaldis `plokid [1, -1, 4]`, mis definitsiooni (55) korral annab välja samuti “1010000”, annab (54) korral tulemuseks osalise listi, sest negatiivse arvuga astendamine tekitab vea.

Veel üks võimalus ülesande lahendamiseks on koostada algul plokkide list ja plokid üheks listiks kokku konkateneerida. Kuna moodulis Prelude on täiesti olemas

operaator `concat`, mille väärtus on listide listi üheks listiks kokkusidumise funktsioon, siis saame selle idee realiseerida koodiga

```
plokid ns
  = concat [show (10 ^ n) | n <- ns, 0 <= n && n < 8]
```

Arvutus viimase definitsiooni järgi on niisama efektiivne kui (55) korral.

Ülesandeid

166. Defineerida muutuja `astmed` väärtuseks funktsioon, mis võtab argumendiks täisarvude listi ja annab tulemuseks listi, mille elemendid saadakse, kui argumentlisti iga mittenegatiivse elemendi n kohta võetakse list esimesest 10 positiivsest täisarvust n-ndas astmes.

167. Modifitseerida ülesande 160 lahendust lokaalse deklaratsiooni abil, et sama numbrite list poleks korduvalt kirjeldatud.

168. Kirjutada avaldis, mille väärtus on kolmas täisruut, mis on suurem kui 100000. Vältida korduvaid arvutusi.

3.3.3 Protseduuride koostamine

Suuremate protseduuride koostamiseks väiksematest saab kasutada teist komprehensioonsüntaksit, monaadikomprehensiooni.

1. Monaadikomprehensioon. Monaadikomprehensioon on sarnane listikomprehensiooniga, kuid võimaldab kirja panna protseduure, liste, nurjumisega tüüpi andmeid ja üldiselt andmeid kõikidest tüüpidest kujul $M\ A$, kus A on tüüp ja M on tüübifunktsioon, mis kuulub klassi Monad. Protseduuride korral M = IO, nurjumisega tüüpide korral M = Maybe, listide korral M = List jne.

Monaadikomprehensioon kannab paralleelnimetust *do*-avaldis võtmesõna **do** järgi, mis teda Haskellis alustab. Üldkuju on

$$\begin{array}{l}\textbf{do}\\ \quad \mathfrak{q}_1\\ \quad \ldots\ ,\\ \quad \mathfrak{q}_l\\ \quad \mathfrak{a}\end{array} \tag{56}$$

kus $l \geqslant 0$ ja iga $\mathfrak{q}_i$ on generaator või lokaalne deklaratsioon ning $\mathfrak{a}$ on avaldis. Generaatorit kujul `_ <- e` võib kirjutada lihtsalt avaldisena $\mathfrak{e}$. Valvureid ei ole, nii et segadust ei teki.

Generaatoritega seotud muutujate nähtavuspiirid on nagu listikomprehensioonis, kusjuures lõpuavaldis $\mathfrak{a}$ vastab selles suhtes listikomprehensiooni püstkriipust vasakul olevale avaldisele. Iga generaatori parema poole avaldis ning samuti avaldis $\mathfrak{a}$ peab olema mingit tüüpi $M\,X$; seejuures M peab olema kõigi nende avaldiste korral sama, aga X võib olla erinev. Kogu *do*-avaldise tüüp langeb kokku avaldise $\mathfrak{a}$ tüübiga.

Me ei hakka siin õppima *do*-avaldise tähendust üldjuhul, kuigi tähenduse ühtne kirjeldus kõigi tüübifunktsioonide M jaoks on põhimõtteliselt võimalik. Vaatame ainult protseduuride juhtu, st juhtu $M = \mathrm{IO}$.

2. *Do*-avaldis protseduuride korral. *Do*-avaldis võimaldab siduda mitu protseduuri kokku üheks suuremaks protseduuriks.

2.1. Põhistsenaarium. Avaldise (56) väärtus on protseduur d, mis täidab järjest kõigi generaatorite paremate poolte väärtuseks olevaid protseduure. Pärast iga sellise alamprotseduuri täitmist kontrollitakse vastava generaatori vasaku poole näidise sobitumist tema edastusväärtusega. Sobitumise korral seotakse $\mathfrak{p}$ sobituselemendid, mittesobitumise korral protseduuri d täitmine katkeb täitmisaegse veaga. Kui kõik generaatorid on edukalt läbitud, täidetakse lõpuks ka $\mathfrak{a}$ väärtuseks olev protseduur ja ühtlasi edastatakse tema edastusväärtus.

Oleme juba tuttavad muutujatega `putStr`, `putStrLn`, `print`, mille väärtuseks on funktsioonid, mis oma argumendil annavad välja teatavaid protseduure. Siia võiks veel lisada muutuja `putChar`, mille väärtuseks on funktsioon, mis võtab argumendiks sümboli ja annab välja protseduuri, mis kirjutab selle sümboli standardväljundisse. Nende operaatorite abil võime *do*-avaldisega kokku kombineerida kompleksse protseduuri, mis teeb mitu sellist tegevust järjest. Näiteks avaldise

```
do
  putStrLn "Tere, Haskell!"
  putChar ' '
  putStr "2 + (-3) = "
  print (2 + (-3))
```
(57)

väärtuseks on protseduur, mis kirjutab ekraanile ühte ritta “Tere, Haskell!” ja teise ritta tühiku järele “2+(-3)=-1”. Kuna väärtusi polnud töö käigus tarvis meelde jätta, võisime generaatorite kohale kirjutada ainult nende paremad pooled.

Üldjuhul nii lihtsalt hakkama ei saa. Vaatleme muutujaid `getChar` ja `getLine`: nende väärtuseks on protseduurid, mis loevad (vajadusel oodates) standardsisendist vastavalt ühe sümboli ja ühe rea (sõnena) ning edastavad loetud info (rea korral ilma reavahetussümbolita). Nüüd näiteks programmi, mis loeb standardsisendist kõigepealt ühe rea ja seejärel üksiku sümboli ning kirjutab siis standardväljundisse teate selle kohta, mis kokku tuli, annab kood

```
main
  = do
      cs <- getLine                                    .   (58)
      c <- getChar
      putStrLn ("\nKokku " ++ cs ++ c : ".")
```

Tõepoolest, *do*-avaldise esimeses reas loetakse rida ja see saab muutuja `cs` väärtuseks, seejärel teises reas loetakse sümbol ja see saab muutuja `c` väärtuseks, lõpuks kolmandas reas kirjutatakse ekraanile uuelt realt sõna "Kokku ", siis `cs` väärtus ehk loetud rida, siis `c` väärtus ehk loetud sümbol ja lõpuks punkt.

Ülesandeid

169. Küsida interaktiivses keskkonnas muutujate `putChar`, `putStr`, `getChar`, `getLine` tüübid ja saada neist aru.

170. Kirjutada programm, mis küsib kasutajalt eesti keeles rida ja kui see tuleb, siis kirjutab ekraanile selle rea sümbolid vastupidises järjekorras.

171. Mida teeb ja mida edastab protseduur, mille kirjeldab avaldis

```
do
  putStr "A\n" ?
  getLine
```

2.2. Erindid. Haskelli standard on erindikäsitluse poolest üsna algeline. Ainsad erindid, mida ta tunneb, on need, mis tekivad protseduuride täitmisel keskkonnaga suheldes. Nende erindite tüüpi märgib tüübikonstruktor `IOError`. (Pangem tähele, et `IOError` ei väljenda protseduuritüüpi.) Hugsi ja GHC laiendused on rikkamad, võimaldades ümber käia ka mujal tekkivate erinditega (kaasa arvatud sisseprogrammeeritud veaolukorrad nagu näidisesobituse ebaõnnestumine ja operaatoriga `error` tekitatavad).

Tavalisimaid tegevusi, kus erindid tekivad, on failidega opereerimine.

Näiteks saab moodulist Prelude failitöötlusoperaatorid `readFile`, `writeFile` ja `appendFile`. Neist `readFile` väärtuseks on funktsioon, mis võtab argumendiks failinime ja annab tulemuseks protseduuri, mis loeb antud nimega faili sisu

ja edastab selle sõnena. Muutujate `writeFile` ja `appendFile` väärtuseks on karritatud funktsioonid, mis võtavad argumendiks failinime ja sõne ja annavad tulemuseks protseduuri, mis kirjutavad selle sõne antud nimega faili; seejuures kui fail on juba olemas, siis `writeFile` puhul tehakse fail enne tühjaks, `appendFile` puhul aga kirjutatakse sõne faili lõppu olemasoleva sisu järele. Kõik need protseduurid heidavad erindi, kui faili ei õnnestunud avada.

(Moodulis Prelude on ainult kõige elementaarsemad failioperatsioonid. Ülejäänud on koondatud moodulitesse System . IO ja System . Directory.)

Do-avaldise väärtuseks oleva protseduuri täitmine toimub vastavalt põhistsenaariumile niikaua, kui generaatori paremate poolte kirjeldatud protseduurid ei heida erindit. Niipea kui mõni neist heidab erindi, lõpetab täitmise ka kogu protseduur, heites sama erindi.

Kui muutuja `main` või mõne interaktiivse keskkonna käsurealt antud avaldise väärtuseks olev protseduur lõpetab täitmise erindiga, tekib täitmisaegne viga.

Vaatame näiteks koodi

```
lugemine
  = do
      putStr "Anna loetava faili nimi. "
      nimi <- getLine
      sisu <- readFile nimi
      putStrLn ("Faili " ++ nimi ++ " sisu:")
      putStrLn sisu
      putStrLn "(Lõpp.)"
```
. (59)

Muutuja `lugemine` väärtuseks saab protseduur, mis küsib faili nime. Kui sisestatud nimega (see satub muutujasse `nimi`) fail leidub ja sel on lugemisõigus, siis loeb protseduur selle faili sisu (see satub muutujasse `sisu`) ja vormistab selle ekraanile. Kui aga faili avamine lugemiseks mingil põhjusel ebaõnnestub, siis heidetakse faili lugemisel erind, mille tagajärjel kogu protseduur lõpetab kohe töö sama erindiga.

3. Mõned kasulikud protseduurid.

3.1. Puhverduse seade. Sisendvoogu lugeda ja väljundvoogu kirjutada on võimalik kas sümbolhaaval või puhverdades. Viimane tähendab sisendvoo puhul, et sealt tulevaid sümboleid ei loeta muidu kui koos puhvritäiega, väljundvoo puhul aga, et sinna kirjutatakse ainult puhvritäite kaupa.

Standardsisendi ja -väljundi puhul esineb tihti soovi puhverdusseadet muuta vastavalt näiteks sellele, kas programm peaks reageerima klahvivajutus-

tele kohe või ootama ka sisestusklahvi vajutamist. Puhverdusseadete muutmiseks on võimalik kirjeldada Haskellis vastav lihtne protseduur ja panna programmis soovitud kohta.

Kahte olulisemat puhverdusseadet, milleks on puhverdamise puudumine ja reakaupa puhverdamine, esindavad konstruktorid `NoBuffering` ja `LineBuffering` tüübist `BufferMode`, mis tulevad koos moodulist System . IO. Standardsisend ja -väljund on antud väärtuseks vastavalt muutujatele `stdin` ja `stdout`, mis tulevad samast moodulist; nende tüüp on `Handle`. Samast saab operaatori `hSetBuffering`, mille väärtus on karritatud funktsioon, mis võtab sisse voopideme ja puhverdusviisi ning annab välja protseduuri, mis seab vastaval vool puhverdusviisi selliseks.

Olemasolevates Haskelli realisatsioonides pole vaikeseade standardvoogude puhul ühesugune ja on isegi versioonist versiooni muutunud.

Näide (58) töötab soovitud viisil vaid siis, kui standardsisendit ei puhverdata. Kui vaikimisi toimub puhverdamine, siis soovitud käitumise saamiseks piisab *do*-avaldise esimeseks reaks lisada avaldis

```
hSetBuffering stdin NoBuffering.
```

3.2. Kaja seade. Kui on soov, et klahvivajutuse puhul ei ilmuks sümbolit ekraanile, siis tuleb keelata standardväljundi kaja. Selleks sobib moodulist System . IO saadav operaator `hSetEcho`, mille väärtus on funktsioon, mis võtab argumendiks voopideme ja tõeväärtuse ja annab välja protseduuri, mis vastavalt tõeväärtusele tekitab või kaotab kaja antud vool.

Proovime näiteks kirjutada programmi, mis ootab ühe sümboli sisestamist klaviatuurilt ja kohe, kui see tuleb, trükib ekraanile selle sümboli koodi. Võiks sobida kood

```
main
  = do
      hSetBuffering stdin NoBuffering .          (60)
      c <- getChar
      print (ord c)
```

Selle programmi käitumine ei pruugi olla ootuspärane, sest ekraanile ilmuvad ka sümbolid ise. Aitab, kui *do*-avaldisse lisada kaja keelav rida

```
hSetEcho stdout False.
```

Sellisel juhul on soovitav lisada sobivasse kohta ka rida, mis kaja uuesti lubab, muidu ollakse sunnitud pärast programmi töö lõppu edasi pimedas töötama.

3.3. Käsureaargumentide lugemine. Käsurea kasutamise tarbeks saab moodulist System . Environment muutuja `getArgs`, mille väärtus on protseduur, mis ei tee muud kui edastab käsureaargumendid sõnede listina.

Näiteks programm, mis toob ekraanile käsureaargumentidest esimese ja viimase, kui neid on vähemalt üks, ja vastasel korral teatab, et argumente pole, võiks välja näha kujul

```
main
  = do
      args <- getArgs
      if null args
        then putStrLn "Argumente pole!"                .
        else
          do
            putStrLn ("Esimene arg.: " ++ head args)
            putStrLn ("Viimane arg.: " ++ last args)
```

3.4. Juhuarvude genereerimine. Tavaliselt me soovime, et genereeritavad juhuarvud poleks programmiga üheselt määratud, vaid erinevatel käivituskordadel ilmuksid erinevad juhuarvud. Seega toob juhuarvu genereerimine funktsionaalse paradigma raames kaasa samad probleemid mis keskkonnast info lugemine (praktikas kasutataksegi juhuslikkuse sissetoomiseks keskkonna infot). Seetõttu pole üllatav, et protseduuritüüpidega tuleb tegemist ka juhuarvudega opereerimisel.

Peamiselt läheb vaja muutujaid `randomIO` ja `randomRIO` moodulist System . Random. Muutuja `randomIO` väärtus on protseduur, mille edastusväärtus võib olla võrdse tõenäosusega ükskõik milline normaalne anne selles tüübis. Tüüp peab olema kontekstist selge või annoteeritud. Muutuja `randomRIO` väärtus on funktsioon, mis võtab argumendiks paari ja annab tulemuseks protseduuri, mille edastusväärtus on võrdse tõenäosusega ükskõik milline anne, mis suuruselt jääb paari komponentidega määratud lõiku. Ka siin võib olla vajalik tüüpi annoteerida. Tüüp, millest juhuandmeid genereeritakse, peab kuuluma klassi Random.

Näiteks avaldise

```
do
  x <- randomRIO (-99 , 99) :: IO Int
  print x
```

väärtus on protseduur, mis genereerib juhuslikult ühe täisarvu, mille absoluutväärtus on ülimalt kahekohaline, ja saadab selle standardväljundisse.

Ülesandeid

172. Küsida interaktiivses keskkonnas `getArgs`, `randomIO`, `randomRIO` tüübid ja saada neist aru.

173. Kirjutada programm, mis küsib kasutajalt eesti keeles sümbolit ja kui see tuleb, kirjutab selle sümboli kohe tühikuga eraldatult küsimuse järele, järgmisele reale aga kirjutab, mitmenda tähega ladina tähestikus tegu on (kui tegu pole tähega ladina tähestikus, siis kirjutab seda).

174. Kirjutada programm, mis teatab ekraanil käsureaargumentide arvu ja kui see oli 2, siis teatab lisaks, kumb argumentidest on tähestikus eespool.

175. Kirjutada programm, mis väljastab ekraanile mittenegatiivse 1-st väiksema arvu täpsusega 4 kohta peale koma, mis oleks võrdse tõenäosusega ükskõik milline selline arv.

176. Kirjutada programm, mis väljastab ekraanile juhuslikult valitud ladina tähestiku tähe, mis võib võrdse tõenäosusega olla üks suurtähtedest ja üks väiketähtedest.

4. Edastusväärtuse seade. Protseduuri edastusväärtust saab ilmutatult seada operaatori `return` abil. Iseseisvalt on muutuja `return` väärtuseks funktsioon, mis suvalisel argumendil x annab tühja, mitte midagi tegeva protseduuri, mis lihtsalt edastab x.

Pangem tähele, et Haskelli `return` ei tähenda tagasipöördumist alamprogrammist nagu paljudes teistes keeltes. Analoogiat on ainult niipalju, et tavaliselt selle muutuja rakendus esineb sarnastes kohtades kus imperatiivsetes keeltes alamprogrammist tagasipöördumise käsk.

Olgu meil näiteks vaja kirjutada protseduur, mis küsib kasutajalt eraldi ridadel kaks täisarvu ja edastab nende summa. Sellise protseduuri defineerib muutuja `küsiSumma` väärtuseks deklaratsioon

```
küsiSumma
  = do
      s <- getLine                          .      (61)
      t <- getLine
      return (read s + read t :: Integer)
```

Siin kasutatud muutuja `read` väärtuseks on funktsioon, mis võtab argumendiks sõne ja annab väärtuseks selle sõne poolt väljendatava täisarvu, ehk sisuliselt parsimine. (Kuna `read` on defineeritud paljude tüüpide jaoks, täpsemalt klassi Read tüüpide jaoks, siis on tüübiannotatsioon vajalik.)

Deklaratsioon (61) võib olla mõttekas, kui kirjutatakse mõnd pikemat programmi, mille erinevates osades on vaja küsida kaks arvu ja leida nende summa. Siis selle asemel, et igale poole vastav kood otse sisse kirjutada, võib kutsuda välja `küsiSumma`. Kuid `küsiSumma` on sellel otstarbel kasutatav ainult juhul, kui küsitud arve endid edaspidi ei vajata, sest `küsiSumma` väärtuseks olev protseduur edastab ainult summa, liidetavad unustab ära.

Muutujat `küsiSumma` kasutavaks näiteprogrammiks sobiks

```
main
  = do
      putStrLn "Anna kaks täisarvu."
      sum <- küsiSumma
      putStrLn ("Summa on  " ++ show sum ++ " .")
```
. (62)

Ülesandeid

177. Kasutades ülesandes 156 defineeritud muutujat `nihe`, defineerida muutuja `niheEdastusel` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab välja protseduuri, mis loeb standardsisendist ühe tähe ja edastab sellest tsüklilises ladina tähestikus n koha võrra tagapool oleva sama "suurusega" tähe (kui loetud sümbol pole täht, siis kirjutab sellesama sümboli).

    Kasutades seda operaatorit, kirjutada programm, mis ootab kasutajalt klahvivajutust ja kui tuleb täht, toob ekraanile hoopis sellest tsüklilises ladina tähestikus järgmise tähe, muidu aga sisestatud sümboli enda.

178. Defineerida muutuja `küsiVahe` väärtuseks protseduur, mis küsib eestikeelse dialoogi abil standardsisendist kaks sümbolit ja edastab nende koodide vahe. Kirjutada programm, mis seda protseduuri kasutades küsib kasutajalt kaks sümbolit ja teatab ekraanil eestikeelse lausega, kas esimene või teine sümbol on kooditabelis teisest ees ja mitme sümboli võrra.

179. Modifitseerida ülesandes 178 kirjutatud protseduuri edastatavat väärtust ja põhiprogrammi tööd nii, et programmi töö lõpus antav teade sisaldaks kasutaja poolt sisestatud sümboleid, mitte poleks juttu vaid "esimesest" ja "teisest".

180. Defineerida muutuja `küsiAlgus` väärtuseks protseduur, mis küsib kasutajalt rea ja edastab sellest reast kuni 10 esimest sümbolit (terve rea, kui seal on vähem sümboleid). Kirjutada programm, mis seda protseduuri kasutades küsib kasutajalt kaks rida ja teatab seejärel ekraanil, millised on nende ridade algusosad, mis meelde jäeti.

5. Erindi seade. Muutujaga `return` tähenduselt analoogilise muutuja `ioError` väärtus on funktsioon, mis võtab argumendiks erindi e ja annab tulemuseks protseduuri, mis midagi ei tee, aga heidab erindi e. Teisalt on `ioError` analoogne operaatoriga `error`, sest ta on kasutatav iga edastusväärtuse tüübi jaoks.

Operaatoriga `ioError` saab programmeerida erindite heitmist oma soovi kohaselt. Selleks peab kirjeldama ka erindi, mida heita; oma erindi loomiseks on operaator `userError`. Tema väärtus on funktsioon, mis võtab argumendiks sõne ja annab tulemuseks erindi, millega seonduv veateade on see sõne.

Näiteks kood

```
failiPea
  = do
      putStr "Anna faili nimi. "
      nimi <- getLine
      sisu@ ~(c : _) <- readFile nimi
      if null sisu
        then ioError (userError "Tühi fail")
        else putStrLn ("Algab " ++ c : "-ga.")
```

annab muutuja `failiPea` väärtuseks protseduuri, mis küsib kasutajalt faili nime ja teatab ekraanil selle faili esimese sümboli — kui aga fail on tühi, siis heidab olukorda iseloomustava veateatega erindi.

Kui loodud oma erind tuleb samas ka seada, siis sobib kasutada operaatorit `fail`, mis on samaväärne operaatorite `userError` ja `ioError` järjestrakendamisega.

Eelmises näites võinuks `ioError (userError "Tühi fail")` asemel kirjutada ka lihtsalt `fail "Tühi fail"`.

Ülesandeid

181. Lasta interaktiivsel keskkonnal näidata muutujate `ioError` ja `userError` tüübid ja saada neist aru.

4 Tsüklilised arvutused

4.1 Rekursioon

Kõik õpitu kvalifitseerub vaid Haskellis programmeerimise sissejuhatuseks, kuni pole omandatud rekursioon — ainus tee meelevaldsete tsükliliste protsesside kirjeldamiseks funktsionaalses paradigmas. Seni kasutasime tsükliliste arvutuste tekitamiseks vaid sisseehitatud erisüntakseid, mis on väga piiratud väljendusvõimsusega.

Rekursiivseks nimetatakse definitsiooni (või definitsioonide rühma), mis paremates pooltes sisaldab defineeritavat objekti ennast (või objektide rühma liikmeid endid). Vastavat nähtust nimetatakse **rekursiooniks**.

Rekursioon on **otsene**, kui ta puudutab korraga vaid üht definitsiooni, vastasel juhul **kaudne**. Teisiti öeldes, kui on antud definitsioonid $\mathfrak{D}_0, \mathfrak{D}_1, \ldots, \mathfrak{D}_{n-1}$, kus iga $i = 0, 1, \ldots, n-1$ korral $\mathfrak{D}_{i+1}$ kasutab midagi, mis on defineeritud definitsioonis $\mathfrak{D}_i$ (kusjuures loeme $\mathfrak{D}_n = \mathfrak{D}_0$), siis $n = 1$ korral on tegu otsese ja $n > 1$ korral kaudse rekursiooniga.

Avaldise väärtustamine rekursiivse definitsiooni või nende rühma järgi tekitab tsüklilise protsessi: kirjutades muutujaid definitsioonide põhjal lahti, võib tulemuses esineda (kas kohe või mingil hilisemal sammul) sama muutuja, mida tuleb siis omakorda asendada. Arvutussammu, kus muutuja asendatakse samast definitsioonist, mille kaudu selle sammuni on jõutud, nimetatakse **rekursiivseks pöördumiseks**. Samamoodi nimetatakse muutuja esinemist oma otseselt rekursiivse definitsiooni paremas pooles.

Efektiivsuse seisukohalt tasub rekursiivseid definitsioone kirjutades jälgida, et tsükliline arvutus sisaldaks ainult samme, mis sõltuvad arvutuse käigus muutuvatest parameetritest. Ülejäänud, muutumatud osad tuleks viia rekursioonist välja, et neid ilmaasjata ei korrataks.

4.1.1 Rekursiivselt defineeritud funktsioonid

Funktsiooni rekursiivsel defineerimisel tüüpiliselt avaldatakse funktsiooni väärtus keerulisemate argumentide korral funktsiooni väärtuste kaudu lihtsamatel argumentidel. Seda üleminekut nimetatakse rekursiooni **sammuks**. Kõige lihtsamatel argumentidel rekursiivset pöördumist ei toimu, tulemus kirjeldatakse nagu rekursioonita definitsioonides. Need juhud moodustavad rekursiooni **baasi**. Keerulisem-lihtsam-seos ja baasjuhud võivad ka puududa; sellisel juhul toimuvad rekursiivsed pöördumised küll piiramatult, kuid näiteks lõpmatu listi arvutamisel see ongi eesmärk.

Keerulisem-lihtsam-seos on erinevatel juhtumitel erinev, seda juba sellepärast, et argumenditüübid on erinevad. Konkreetse rekursiivse definitsiooni väljamõtlemiseks peab programmeerija otsustama, millist suurusseost kasutada, kuidas toimub taandamine väiksematele juhtudele ning millised on baasjuhud ja kuidas neid lahendada.

1. Rekursioon täisarvudel. Täisarvulise argumendi puhul võib keerulisem-lihtsam-seos kokku langeda arvude suurusjärjestusega.

1.1. Tüüpilised olukorrad. Tüüpiliselt on üks baasjuht — argument 0 — või on lisaks kõiki negatiivseid arve käsitlev baasjuht. Vastavalt tuleb rekursiivses definitsioonis kirjeldada, kuidas funktsiooni väärtus suuremal arvul avaldub funktsiooni väärtuse kaudu väiksemal (enamasti 1 võrra) arvul ning mis on funktsiooni väärtus kohal 0. Tihti esineb baasjuhuna ka 1 või veel mõni muu arv.

Rekursiivse definitsiooni tuletamise esimese näitena olgu meil vaja defineerida muutuja `suurSumma` väärtuseks funktsioon, mis võtab argumendiks naturaalarvu n ja annab tulemuseks arvu $1^4 + \ldots + n^4$ ehk 1-st n-ni kõigi täisarvude 4. astmete summa. Tüübisignatuuriks sobib

```
suurSumma
  :: (Integral a).
  => a -> a
```

Paneme tähele, et kui $n > 0$ ja meil on funktsiooni väärtus kohal $n-1$ ehk $1^4 + \ldots + (n-1)^4$ juba teada, siis leidmaks funktsiooni väärtust kohal n, piisab funktsiooni väärtusele kohal $n-1$ liita arv n^4. Kui aga $n = 0$, siis summas liidetavaid pole, mistõttu funktsiooni väärtus on 0. Võttes mängu ka negatiivse argumendi võimaluse, mispuhul arvutus võiks lõpetada töö informatiivse veateatega, saame

koodi

```
suurSumma n
  = case compare n 0 of
      GT
        -> suurSumma (n - 1) + n ^ 4
      EQ
        -> 0
      _
        -> error "suurSumma: neg. liidetavate arv"
```
. (63)

Iga funktsioon, mis antakse argumendil n kui summa mingi teise funktsiooni väärtustest argumentidel 1-st n-ni, on sarnasel viisil rekursiivselt defineeritav. See tuleneb asjaolust, et summaoperaator põhimõtteliselt on matemaatiliselt rekurrentselt defineeritav. Seejuures argumendil 0 on funktsiooni väärtus alati 0, sest 0 on liitmise ühikelement (ei muuda summat, kui ta lisada). Analoogne jutt kehtib, kui summa asemel rääkida korrutisest, ainult et argumendil 0 on sellise funktsiooni väärtus 1.

Rekursiooni sammul võib olla vaja muuta mitut parameetrit.

Vaatame näiteks diskreetsest matemaatikast tuntud **kahaneva faktoriaali** funktsiooni

$$(x)_k = x \cdot (x-1) \cdot \ldots \cdot (x-(k-1)), \tag{64}$$

millel on kaks argumenti: arv x ja naturaalarv k. Kui $k > 0$, siis saab seose (64) paremas pooles eraldada esimese teguri x, ülejäänud tegurite korrutis on $(x-1)_{k-1}$. Juhul $k = 0$ on tegurite arv 0, mistõttu korrutis on 1. Neil tähelepanekutel põhinev kood kahaneva faktoriaali arvutamiseks on

```
kahFac x k
  = case compare k 0 of
      GT
        -> x * kahFac (x - 1) (k - 1)
      EQ
        -> 1
      _
        -> error "kahFac: neg. teine argument"
```
. (65)

Et esimene argument ja ühtlasi korrutis võib olla suvalist tüüpi arv, kuid tegurite arv peab olema täisarv, siis tüübisignatuuriks sobib

```
kahFac
  :: (Num a, Integral b).
  => a -> b -> a
```

Ülesandeid

182. Arvu n **faktoriaal** $n!$ avaldub kahaneva faktoriaali kaudu seosega $n! = (n)_n$. Seetõttu saab definitsiooniga (65) antud muutujat `kahFac` kasutades kodeerida faktoriaalifunktsiooni definitsiooniga

```
fac n
  = kahFac n n
```

Anda muutujale `fac` samaväärne rekursiivne definitsioon ilma muutujat `kahFac` kasutamata.

183. Kirjutada muutujale `kahFac` koodiga (65) samaväärne definitsioon, kus rekursioon põhineks võrduse (64) viimase teguri eraldamisel.

184. Matemaatikas defineeritakse **kasvava faktoriaali** funktsioon võrdusega

$$(x)^k = x \cdot (x+1) \cdot \ldots \cdot (x+(k-1)).$$

Defineerida muutuja `kasFac` väärtuseks kasvava faktoriaali funktsiooni karritamise tulemus.

185. Defineerida muutuja `maxMod` väärtuseks karritatud funktsioon, mis võtab argumendiks täisarvud m ja n: kui m on positiivne ja n mittenegatiivne, siis annab väärtuseks maksimaalse jäägi, mis arvude 1 kuni n ruutude jagamisel m-ga tekib ($n = 0$ korral 0); vastasel korral lõpetab arvutus töö veateatega, mis ütleb, kumb muutuja nõutud tingimustele ei vastanud.

1.2. Lihtsaim pole alati efektiivne. Rekursiivne definitsioon on tihti otseselt tuletatav matemaatikas kasutatavast rekurrentsest võrrandist. Kuigi Haskell on väljendusvahendina küllalt võimas, et tõlkida sellesse matemaatikas antavaid definitsioone, pole matemaatikas standardse definitsiooni otsene ülevõtmine alati rahuldav, kuna annaks liiga aeglase arvutuse. Programmeerimisel tuleb definitsiooni kirjutamisel ka arvutusefektiivsust silmas pidada.

Võtame näiteks **Fibonacci** jada

$$0 \quad 1 \quad 1 \quad 2 \quad 3 \quad 5 \quad 8 \quad 13 \quad 21 \quad 34 \quad 55 \quad 89 \quad 144 \quad 233 \quad 377 \quad 610 \quad 987 \quad \ldots,$$

mille esimesed elemendid on 0 ja 1 ja ülejäänutest igaüks leitakse kui kahe eelmise summa. Matemaatikas pannakse see reegel kirja seostega

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall n \geqslant 2\,(F_n = F_{n-1} + F_{n-2}) \tag{66}$$

ning rekurrentse seose $F_n = F_{n-1} + F_{n-2}$ kehtivuse laiendamiseks juhule $n < 2$ lisatakse definitsioonile negatiivseid indekseid puudutav klausel

$$\forall n < 0 \left(F_n = (-1)^{n+1} F_{-n} \right).$$

Nii saab F kõigil täisarvudel defineeritud täisarvuliste väärtustega funktsiooniks.

Kui tõlgiksime selle definitsiooni naiivselt Haskelli, võiksime saada Fibonacci jada liikmete ehk Fibonacci arvude arvutamiseks näiteks deklaratsiooni

```
fib n
  | n >= 2
    = fib (n - 1) + fib (n - 2)
  | n >= 0
    = n
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
, (67)

tüübisignatuur võib olla

```
fib
  :: (Integral a).
  => a -> a
```

Kasutatud muutuja `odd` väärtus on predikaat, mis annab välja True parajasti siis, kui argument on paaritu arv. On olemas ka muutuja `even`, mille väärtus on predikaat, mis annab True parajasti siis, kui argument on paarisarv.

Koodi (67) järgi arvutades toob iga `fib` väljakutse 1-st suurema väärtusega argumendil kaasa sama operaatori kaks uut väljakutset. Et kumbagi töödeldakse teisest sõltumatult, kasvab väljakutsete arv eksponentsiaalselt. Arvutus on seetõttu nii aeglane, et kuigi palju Fibonacci arve koodiga (67) reaalselt võimalik arvutada ei ole.

1.3. Opereerimine mitme vahetulemusega. Kui funktsiooni väärtuse arvutamisel on vaja opereerida mitme rekursiivselt arvutatud andmega, siis mitme rekursiivse pöördumise vältimiseks on kasulik rekursiivselt defineerida hoopis funktsioon, mis annaks vajalikud andmed koos välja struktuurina.

Loomulikult on võimalik Fibonacci arve arvutada järjekorranumbriga võrdelise liitmiste arvuga. Selleks oleks vaja, et definitsioon piirduks ühe rekursiivse pöördumisega. See oleks võimalik, kui kaks viimast vahetulemust oleks koos kättesaadavad.

Otseseim võimalus selleks on muuta definitsiooni nii, et defineeritav funktsioon annaks välja paare kahest järjestikusest Fibonacci arvust. Siis iga selline paar (F_n, F_{n+1}) on leitav eelmise sellise paari (F_{n-1}, F_n) kaudu, nihutades teise komponendi esimeseks ja pannes teiseks komponendiks komponentide summa. Õige

funktsiooni väärtus avalduks siis kui samal argumendil leitud abifunktsiooni väärtuse üks komponent.

Defineerime abifunktsiooni originaalfunktsiooni definitsiooni kehas lokaalse muutuja väärtuseks. Kasutades lokaalse muutuja jaoks lihtsuse mõttes sama nime `fib` (mis `fib` välise operaatori tähenduses lokaalselt varjutab), saame definitsiooniks

```
fib n
  | n >= 0
    = let
        fib 0
          = (0 , 1)
        fib n
          = let
              (u , v)                          .      (68)
                = fib (n - 1)
            in
            (v , u + v)
      in
      fst (fib n)
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```

Paari avaldamiseks eelmise kaudu on omakorda kasutatud lokaalset deklaratsiooni, mis seob muutujad `u` ja `v` eelmise paari komponentidega. Nii pääseme operaatorite `fst` ja `snd` korduvast rakendamisest.

Trikk defineeritava muutuja nime varjutamisega lokaalselt läheb läbi tänu sellele, et lokaalse `fib` nähtavuspiirkonnas pole põhioperaatorit `fib` vaja välja kutsuda. Globaalses tähenduses on `fib` kasutatud ainult kahes kohas: kohe koodi alguses ja lõpu eel rakendatuna argumendile `-n`. Kõik teised kasutused on lokaalses tähenduses.

Teise näitena paaride kasutamisest võtame olukorra, kus meile pakub huvi kaherealine lõpmatu tabel

$$\begin{array}{cccccccccccc} 0 & 1 & 2 & 5 & 12 & 29 & 70 & 169 & 408 & 985 & 2378 & 5741 \\ 1 & 1 & 3 & 7 & 17 & 41 & 99 & 239 & 577 & 1393 & 3363 & 8119 \end{array} \ldots, \quad (69)$$

mille esimeses veerus on 0 ja 1 ning iga ülejäänud veeru ülemine arv on eelmise veeru arvude summa ja alumine arv on selle ja eelmise veeru ülemiste arvude summa.

Selle tabeli arvude arvutamiseks kujutame tabelit ette paaride jadana, kus paarid vastavad tabeli veergudele. Jada algab paariga $(0, 1)$ ja iga ülejäänud liige on võimalik esitada eelmise kaudu (teise komponendi esitamiseks kasutame esimese komponendi esitust, st ta avaldub kui eelmise paari esimese komponendi ja komponentide

summa summa). Kirjeldame selle ülemineku koodiga

```
järgmVeerg
  :: (Integral a)       ,
  => (a , a) -> (a , a)

järgmVeerg (u , v)
  = let
      s = u + v    .
    in
    (s , u + s)
```
(70)

Nüüd defineerime muutuja `sq2` väärtuseks funktsiooni, mis naturaalarvulisel argumendil n annab välja selle jada liikme nr n (nummerdades liikmeid alates 0-st). Tüübisignatuuriks sobib

```
sq2
  :: (Integral a).
  => a -> (a , a)
```

Üldistades leitud rekurrentset seost ka negatiivses suunas, saame definitsiooni

```
sq2 n
  = case compare n 0 of
      GT
        -> järgmVeerg (sq2 (n - 1))
      EQ
        -> (0 , 1)
      _
        -> let
             (a , b)
               = sq2 (-n)
           in
           if even n then (-a , b) else (a , -b)
```
. (71)

Arvutuse maht on lineaarne, st võrdeline paari järjekorranumbriga. (Nende paaride komponentide suhe lähendab arvu $\sqrt{2}$, sellest ka muutuja nime valik.)

Ülesandeid

186. Lucas' jada defineeritakse rekurrenteselt seostega

$$L_0 = 2, \qquad L_1 = 1, \qquad \forall n \geqslant 2\,(L_n = L_{n-1} + L_{n-2}).$$

Negatiivsetel indeksitel võib seda täiendada reegliga $L_n = (-1)^n \cdot L_{-n}$.

Defineerida muutuja `luc` väärtuseks funktsioon, mis täisarvulisel argumendil n annab väärtuseks L_n. Arvutuse maht peab kasvama argumendi suhtes lineaarselt.

2. Struktuurne rekursioon listidel. Struktuurne rekursioon tähendab, et funktsiooni väärtus suurematel andmestruktuuridel määratletakse funktsiooni väärtuste kaudu väiksematel andmestruktuuridel (suurema osadel). Andmestruktuuriks on väga tihti just listid.

Kõik funktsioonid listidel, mis teevad midagi määramata arvu elementidega, tuleb programmeerida rekursiooniga. Muidugi on palju kasulikke arvutusi võimalik realiseerida eeldefineeritud muutujate abil, aritmeetilise jada süntaksiga või komprehensioonsüntaksiga, kuid sellised realisatsioonid pole alati kõige efektiivsemad.

2.1. Tüüpilised olukorrad. Listidel töötava funktsiooni definitsioon tüüpiliselt väljendab funktsiooni väärtuse mittetühjal listil funktsiooni väärtuse kaudu selle listi sabal ning tüüpilise baasjuhuna esineb tühi list, kuid tihti läheb vaja ka keerulisemaid rekursiooniskeeme.

Olgu meil vaja arvulisti järgi leida tema nulliga võrdsete elementide arv. Listikomprehensioon võimaldab lühikesi ja elegantseid teid selleks, nagu näiteks `length [x | x <- l, x == 0]` ja `length [0 | 0 <- l]`, kus `l` on avaldis, mille väärtus on meid huvitav list. Sellise koodi järgi arvutades luuakse originaallisti nullidest omaette list ja loetakse siis sealt need nullid kokku.

Rekursiooniga on võimalik saada efektiivsem arvutus, mis ei tekita mõttetut nullide vahelisti, vaid loendab neid originaallisti peal. Vajaliku funktsiooni signatuur olgu

```
nullideArv
  :: (Num a)    .
  => [a] -> Int
```

Rekursiooniskeemi koostamiseks oletame, et mittetühja listi saba nullide arv on teada. Kui nüüd listi pea on null, siis kogu listis on nulle ühe võrra rohkem kui sabas, ülejäänud juhtudel on kogu listis niisama palju nulle kui tema sabas. Rekursiooni baasiks on vaja ka tühja listi nullide arvu, mis on 0. Saame definitsiooni

```
nullideArv (x : xs)
  = let
      ülejäänu
        = nullideArv xs
    in                                                    .   (72)
    if x == 0 then 1 + ülejäänu else ülejäänu
nullideArv _
  = 0
```

Teise näitena vaatame kaht sarnast ülesannet: esimeses on vaja leida antud listis kõik teatud suurusest väiksemad elemendid ja moodustada neist omaette list; teises

tuleb omaette listi panna kõik need elemendid, mis esimeses välja jäävad. Suurust, mis sellises ülesandes esineb eralduspiirina, nimetatakse lahkmeks.

Võiks teha kummagi listi eraldamiseks oma operaatori. Kuna aga nende töö oleks suures osas ühine — elemendid jaotataks sama lahkme järgi kahte rühma, lihtsalt ühe puhul antaks lõpus välja üks rühm ja teise puhul teine —, siis on otstarbekas kirjutada üks operaator, mis läbiks argumentlisti ühe korra, koostaks mõlemad vajalikud listid korraga ja tagastaks nad koos. (Sarnasel põhjusel on olemas eeldefineeritud operaatorid `splitAt` jt, millest on eelnevalt juttu olnud.)

Täpsemalt, defineerime muutuja `kaheksLahuta` väärtuseks funktsiooni, mis võtab argumendiks andme x suvalisest järjestusega tüübist ja sama tüüpi andmete listi l ning annab tulemuseks paari kahest listist, kus ühes on listi l need elemendid, mis on x-st väiksemad, ja teises ülejäänud. Signatuuriks on

```
kaheksLahuta
  :: (Ord a)                    ,
  => a -> [a] -> ([a] , [a])
```

definitsiooniks sobib

```
kaheksLahuta x (z : zs)
  = let
      (us , vs)
        = kaheksLahuta x zs
    in
    if z < x                    .
      then (z : us , vs)
      else (us , z : vs)
kaheksLahuta _ _
  = ([] , [])
```
(73)

Tõepoolest, mittetühja listi esimene element paigutatakse kas paari esimesse või teise komponenti vastavalt sellele, kas ta on lahkmest väiksem või mitte, ja ülejäänud elemendid jaotatakse samal viisil. Tühja listi jaotus annab kaks tühja listi.

Kui mõni argument antakse rekursiivsel pöördumisel alati edasi samasugusena nagu ta sisse tuli, siis saab selle argumendi rekursioonist välja viia, defineerides rekursiivselt hoopis abimuutuja, millel seda argumenti pole.

Definitsioonis (73) antakse esimene argument rekursiivsel pöördumisel alati muutmata kujul edasi: ainus rekursiivne väljakutse asub *let*-plokis ja argument seal on `x`, mis langeb kokku argumendiga deklaratsiooni vasakus pooles. Viies rekursiooni

lokaalsesse deklaratsiooni, saame definitsiooni

```
kaheksLahuta x xs
  = let
      lahuta (z : zs)
        = let
            (us , vs)
              = lahuta zs
          in
          if z < x
            then (z : us , vs)
            else (us , z : vs)
      lahuta _
        = ([] , [])
    in
    lahuta xs
```
, (74)

milles rekursioon muutumatu väärtusega argumenti kaasas ei kanna. Selline optimisatsioon parandab arvutuskiirust siiski vaid marginaalselt.

Ülesandeid

187. Otsida Hugsi teegist muutujate `!!`, `take`, `drop`, `splitAt` definitsioonid ja saada neist aru.

188. Defineerida muutuja `suurtähega` väärtuseks funktsioon, mis võtab argumendiks sõnede listi ja annab välja suurtähega algavate sõnede arvu selles.

189. Defineerida muutuja `esitähed` väärtuseks funktsioon, mis võtab argumendiks sõnede listi ja annab tulemuseks nende sõnede algusosadest moodustatud lühendi. Sõnedest, kus on vähemalt kaks tähte, peab lühendisse tulema parajasti kaks esimest tähte; ühetähelistest sõnedest peab tulema nende ainus täht; tühjad sõned annavad lühendisse tühiku.

190. Otsese rekursiooniga anda muutuja `eemaldaEsim` väärtuseks karritatud funktsioon, mis võtab sisse võrdusega tüüpi andme x ja sama tüüpi andmete listi l ning annab välja paari, kus esimene komponent on True või False vastavalt sellele, kas x esineb listis l või mitte, ja teine komponent on list, mis saadakse l-st esimese x esinemise eemaldamisel (või l ise, kui sellist pole).

191. Anda muutuja `eemaldaKõik` väärtuseks karritatud funktsioon, mis võtab samad argumendid x ja l nagu ülesandes 190 kirjeldatud funktsioon ja annab välja paari, mille esimene komponent on nagu tolle funktsiooni puhul, kuid teine komponent saadakse listist l kõigi x esinemiste eemaldamisel.

192. Defineerida muutuja `eemaldaJaLoenda` väärtuseks karritatud funktsioon, mis võtab samasugused argumendid x ja l nagu ülesande 191 funktsioon ning annab väärtuseks paari, mille esimene komponent on x esinemiste arv listis l ja teine komponent on list, mis saadakse l-st kõigi x esinemiste eemaldamisel. Minimiseerida korduvat tööd arvutuse käigus.

193. Defineerida muutuja `minEtte` väärtuseks funktsioon, mis argumendiks saadud lõplikul mittetühjal järjestusega tüüpi elementide listil l annab tulemuseks listi, mis saadakse l-st tema minimaalse elemendi ümbertõstmisel algusse (kui minimaalseid on mitu, siis tõstetakse esimene selline).

194. Defineerida muutuja `korrutisteSumma` väärtuseks funktsioon, mis võtab argumendiks arvulisti ja annab tulemuseks tema elementide paarikaupa korrutiste summa.

195. Defineerida muutuja `võrdlePikkust` väärtuseks karritatud funktsioon, mis võtab argumendiks kaks listi ja annab tulemuseks nende pikkuste võrdlemise tulemuse tüübi Ordering väärtusena. Juhul, kui üks listidest on lõpmatu, teine lõplik, tuleb lõpmatu list tituleerida pikemaks.

2.2. Mitme baasjuhuga rekursioon. Mõnikord ei piisa sellest, kui programmeeritakse rekursiooni baas vaid tühja listi jaoks. Tihti tuleb seda ette siis, kui rekursiooni sammul läheb vaja listi mitut elementi; siis näiteks üheelemendilise listi juht tuleb eraldi programmeerida.

Olgu meil näiteks vaja defineerida muutuja `kahesummad` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja leiab tema kõikvõimalike kahest järjestikusest kohast võetud elementide summade listi; signatuuriks sobib

```
kahesummad
  :: (Num a)    .
  => [a] -> [a]
```

Et ühtki summat leida, peab olema vähemalt kaks elementi olemas, seetõttu on tühja ja üheelemendilise listi juhud vaja käsitleda üldjuhust eraldi. Sobib definitsioon

```
kahesummad (x : xs@ (y : _))
  = x + y : kahesummad xs
kahesummad _                  .
  = []
```
(75)

Esimene deklaratsioon vastab juhule, kus argumentlistis on vähemalt kaks elementi, sest listi saba näidises on sealne esimene element eraldi välja toodud. Kuna funktsiooni väärtus tühjal ja üheelemendilisel listil on võrdne tühja listiga (ühtki summat pole), õnnestub need kaks juhtu teises deklaratsioonis ühekorraga ära kirjeldada ja piirduda kokku kahe deklaratsiooniga.

Ülesandeid

196. Defineerida kahel viisil muutuja `kõikVõrdsed` väärtuseks predikaat, mis võtab argumendiks listi ja kontrollib, kas listi elemendid on kõik võrdsed. Kui pole, siis peab väärtuseks tulema False, vastasel korral kui list on lõplik, siis peab väärtuseks tulema True. Esimene definitsioon peab realiseerima idee kontrollida järjestikuste elementide võrdust, teine aga idee kontrollida esimese elemendi võrdumist ülejäänutega.

197. Defineerida muutuja `kuniKorduseni` väärtuseks funktsioon, mis võtab argumendiks listi ja annab välja selle listi algusosa kuni esimese elemendini, mis võrdub talle järgneva elemendiga, kui selline koht listis leidub, vastasel korral annab välja argumentlisti enda.

198. Defineerida muutuja `kordaViimast` väärtuseks funktsioon, mis võtab argumendiks listi: kui see pole tühi ega lõpmatu, siis annab tulemuseks listi, kus on järjest argumentlisti elemendid ja viimane veel korra; lõpmatu listi puhul annab tulemuseks argumentlisti enda; tühja listi puhul lõpetab arvutus töö etteantud sobiva veateatega.

199. Defineerida muutuja `needi` väärtuseks karritatud funktsioon, mis võtab argumendiks kaks listi: kui esimene element lõpeb sama elemendiga, millega teine algab, siis annab tulemuseks listi, kus argumentlistid on konkateneeritud, kuid leitud ühine element esineb ühekordselt; kui esimese listi viimane ja teise esimene element pole võrdsed, või kui esimene list on tühi või lõpmatu, siis annab tulemuseks esimese listi.

200. Defineerida muutuja `sobita` väärtuseks karritatud funktsioon, mis võtab argumentideks sõned r ja s ja annab välja tõeväärtuse vastavalt sellele, kas regulaaravaldisena interpreteeritud r sobitub sõnega s. Regulaaravaldises on sümbolid ? ja * eritähenduslikud, mida võib asendada vastavalt ühe suvalise sümboliga ja suvalise lõpliku sümbolijadaga.

2.3. Ebastandardne rekursioon. Mõnikord tuleb kasutada rekursiivset pöördumist mitte listi sabal, vaid saba sabal või mõnel veel sügavamal alamstruktuuril.

Olgu meil vaja defineerida muutuja `kaheksLoe` väärtuseks funktsioon, mis võtab argumendiks listi l ja annab tulemuseks listipaari, kus l elemendid on üle ühe jaotatud kahte listi; signatuur olgu

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

Sellise jaotamise tegemisel tuleb igal sammul teada, kummasse listi käsilolev element paigutada. Põhiprobleemiks on, kuidas seda infot hoida.

Loomulik lahendus on töödelda rekursiooni sammul kaks elementi. Sellisel juhul on kindel, et esimene element peab minema esimesse listi. Et rekursiooni samm lühendab listi 2 elemendi võrra, peab sammu tegemiseks 2 elementi olemas olema ja nii tühi kui ka 1-elemendilised listid jäävad baasjuhtudeks. Kuid need baasjuhud on kirjeldatavad ühtsel viisil. Nii saame definitsiooni

```
kaheksLoe (x : y : ys)
  = let
      (us , vs)
        = kaheksLoe ys
    in
    (x : us , y : vs)
kaheksLoe xs
  = (xs , [])
```
. (76)

Siin on siiski ka elegantsem lahendus, mis töötleb korraga ühe elemendi. Idee on vahetada igal sammul ehitatava paari komponendid. Selle idee realiseerib kood

```
kaheksLoe (x : xs)
  = let
      (us , vs)
        = kaheksLoe xs
    in
    (x : vs , us)
kaheksLoe _
  = ([] , [])
```
. (77)

Niisugune lahendus on võimalik tänu sellele, et rekursiooni baas on kahe variandi suhtes sümmeetriline: me ei pea seal teadma, kumb komponent saab lõpuks olema vasak ja kumb parem. Me tuleme hiljem tagasi sarnase näite juurde, mis seda lahendust ei võimalda. Teine soodne asjaolu on muidugi see, et lisanduv operatsioon — paari komponentide vahetus — tuleb siin tasuta kätte.

Rekursiivse pöördumise argumendi valik võib ka listist sõltuda, nii et programmeerija ei saa konkreetse hüppe suurusega arvestada.

Defineerime muutuja `segmendid` väärtuseks funktsiooni, mis argumendiks saadud listi järgi koostab tema segmentide listi, kus listi segmendiks nimetame maksimaalseid mittekahanevaid lõike listis (st mittekahanevas järjestuses elementidega lõike, mis ei sisaldu oma kohal üheski pikemas sellises lõigus). Esitades iga segmendi omaette listina, peab tulemuslist koosnema listidest ehk definitsioon peab

rahuldama signatuuri

```
segmendid
  :: (Ord a)      .
  => [a] -> [[a]]
```

Kõigepealt programmeerime esimese segmendi eraldamise. Kood

```
eraldaSegment
  :: (Ord a)           ,
  => [a] -> ([a] , [a])

eraldaSegment (x : xs@ ~(y : _))
  | null xs || x > y
    = ([x] , xs)
  | otherwise
    = let
        (us , vs)                                    (78)
          = eraldaSegment xs
      in
      (x : us , vs)
eraldaSegment _
  = ([] , [])
```

annab muutuja `eraldaSegment` väärtuseks funktsiooni, mis võtab argumendiks listi ja annab välja tema esimese segmendi paaris ülejäänud elementide listiga. Esimeses valvuris on kahanemise juht kokku võetud juhuga, kus listi saba on tühi, sest mõlemal juhul otsitav segment lõpeb kohe pärast listi pead ja väljund on sarnane.

Kasutades seda muutujat, saab nüüd defineerida muutuja `segmendid` koodiga

```
segmendid xs
  | null xs
    = []
  | otherwise
    = let                      .                 (79)
        (us , vs)
          = eraldaSegment xs
      in
      us : segmendid vs
```

Definitsioon (79) on küll rekursiivne, sest teise valvuri juht sisaldab pöördumist `segments` poole, kuid rekursiooniskeem on ebatüüpiline, kuna rekursiivsel pöördumisel ei ole argumendiks listi saba, vaid teise operaatori abil arvutatud alamlist.

Ülesandeid

201. Kirjutada oma moodulisse definitsioon (76) või (77) ja veenduda interaktiivses keskkonnas, et defineeritud muutuja `kaheksLoe` rakendamisel lõpmatule argumentlistile on tulemuseks paar kahest lõpmatust listist.

202. Defineerida muutuja `vahelduv` väärtuseks funktsioon, mis võtab argumendiks sõne ja annab väärtuseks sõne, mille igal paarisarvulise indeksiga kohal on algse sõne vastaval kohal oleva tähe suurtäheline variant ning igal paarituarvulisel kohal algse sõne vastaval kohal oleva tähe väiketäheline variant.

203. Defineerida muutuja `altern` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab tulemuseks tema elementide alternatiivse summa, st üle ühe on elemendid liidetud ja lahutatud.

204. Defineerida muutuja `paarisNegArv` väärtuseks funktsioon, mis leiab argumentlisti paariskohtadel (st indeksiga $0, 2, \ldots$) olevate negatiivsete elementide arvu.

205. Kasutades ülesandes 191 defineeritud muutujat `eemaldaKõik`, defineerida muutuja `korduvad` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks paarikaupa erinevate elementidega listi, mis koosneb parajasti argumentlistis korduvatest elementidest.

206. Defineerida muutuja `läbivSuur` väärtuseks funktsioon, mis võtab argumendiks sõne ja annab tulemuseks sõne, kus võrreldes argumentsõnega on sõne alustava väiketähe (kui sõne algab väiketähega) ja iga tühisümbolile järgneva väiketähe kohal vastav suurtäht.

207. Kas definitsioonis (79) esimese valvuri juhu ärajätmisel saame samaväärse definitsiooni?

208. Kas definitsioonis (79) esimese valvuri juhus `[]` asendamisel `xs`-ga saame samaväärse definitsiooni?

3. Vastastikrekursioon. Kaudset rekursiooni, kus kaks definitsiooni kasutavad paremas pooles kumbki teise defineeritavat objekti, nimetatakse vastastikrekursiooniks.

Lisaks definitsioonidele (76) ja (77) saab muutuja `kaheksLoe` samaväärselt defineerida vastastikrekursiooniga. Selleks kirjutame kaks definitsiooni, millest üks on elemendi paigutamiseks vasakpoolsesse, teine aga parempoolsesse listi. Saame

koodi

```
kaheksLoe (x : xs)
  = let
      (us , vs)
        = kaheksLoe' xs
    in
    (x : us , vs)
kaheksLoe _
  = ([] , [])
```

.

(80)

```
kaheksLoe' (x : xs)
  = let
      (us , vs)
        = kaheksLoe xs
    in
    (us , x : vs)
kaheksLoe' _
  = ([] , [])
```

Muutuja `kaheksLoe` definitsioon kasutab muutujat `kaheksLoe'`, mille definitsioon omakorda kasutab muutujat `kaheksLoe`. Seega on need definitsioonid vastastikku rekursiivsed.

Kuna meile pakub huvi ainult muutuja `kaheksLoe` ja muutuja `kaheksLoe'` ainus väljakutse asub `kaheksLoe` definitsiooni kehas, võib `kaheksLoe'` definitsiooni muuta lokaalseks, viies ta üle `kaheksLoe` definitsiooni *let*-plokki.

Ülesandeid

209. Lahendada ülesanded 202, 203, 204 vastastikrekursiooniga.

210. Defineerida kolmel viisil erinevate tehnikatega muutuja `vaheliti` väärtuseks karritatud funktsioon, mis võtab argumendiks kaks listi ja koostab vastuseks listi, mille elemendid tulevad vaheldumisi ühest ja teisest listist, niikaua kui võtta saab. Kui ühe listi elemendid lõpevad, siis teise listi ülejääv osa jääb muutmata kujul vastuslisti lõppu.

4.1.2 Rekursiivselt defineeritud protseduurid

Tõsiasi, et tsükliliste arvutuste programmeerimiseks pole muud valikut kui kasutada rekursiooni, kehtib ka tsüklis jooksva interaktiivse protsessi kohta.

1. Definitsioonid läbi funktsiooni. Interaktiivseid programme saab kirjutada tavalise funktsioonirekursiooniga, kus defineeritava funktsiooni väärtuseks on protseduurid.

Kirjutame näitena programmi, mis harjutab kasutajat klaviatuuril vigadeta tippima. Programm peab esitama ekraanil kasutajale tekstiridu, mida kasutaja peab järele kirjutama. Niipea, kui kasutaja mõne tähega eksib, esitab programm sama rea uuesti. Kui kasutaja viga ei tee, jätkab programm järgmise reaga või, kui kogu tekst on läbi, lõpetab.

Jagame koodi osadeks järgnevalt.

Muutuja `kordaTähed` väärtuseks defineerime funktsiooni, mis võtab argumendiks ühe rea ja annab tulemuseks protseduuri, mis ootab standardsisendist selle rea kordamist ja edastab tõeväärtuse vastavalt sellele, kas kordamine õnnestus või mitte. See protseduur ise ei tegele rea esitamisega kasutajale, selle eest peab hoolitsema väljakutsuv protseduur.

Muutuja `kordaRead` väärtuseks olgu funktsioon, mis võtab argumendiks ridade listi ja annab tulemuseks protseduuri, mis esitab neid ridu kasutajale kordamiseks. Tema definitsioon peab kutsuma välja operaatorit `kordaTähed` ja valima järgnevalt esitatava rea sellelt saadud tõeväärtuse järgi.

Põhiprogramm, mis kutsub välja `kordaRead`, olgu muutujas `harjutus`.

Signatuurid on selle spetsifikatsiooni põhjal

```
kordaTähed
  :: String -> IO Bool'

kordaRead
  :: [String] -> IO ()'

harjutus
  :: IO ()
```

Definitsioonide kirjutamist alustame lihtsamast ehk põhiprogrammist. Sobib kood

```
harjutus
  = do
      hSetBuffering stdin NoBuffering     .      (81)
      kordaRead tekst
      putStrLn "Harjutus lõpuni läbitud."
```

Operaatori `kordaRead` argument `tekst` peab väärtuseks saama harjutuse teksti (ridade listina), mille anname praegu lihtsuse mõttes koodis ette definitsiooniga

```
tekst
  = words "gladiool hüatsint krüsanteem liilia"      (82)
```

Operaator `words` tuleb moodulist Prelude ja tema väärtuseks on funktsioon, mis annab teksti järgi välja tema sõnade listi vastavalt tühisümbolite esinemisele tekstis. Kõrvalepõikena võib märkida, et moodul Prelude pakub ka analoogilise operaatori `lines`, mis annab teksti järgi välja ridade listi vastavalt reavahetussümbolite esinemisele. Seega deklaratsiooniga (82) samaväärne oleks deklaratsioon

```
tekst
  = lines "gladiool\nhüatsint\nkrüsanteem\nliilia"
```

Nüüd jõuame tsüklite kirjutamiseni. Muutuja `kordaRead` definitsiooniks sobib

```
kordaRead rs@ (w : ws)
  = do
      putStrLn w
      b <- kordaTähed w
      putStrLn ""
      if b then kordaRead ws else kordaRead rs
kordaRead _
  = return ()
```

. (83)

Tõepoolest, mittetühja argumentlisti korral kirjutatakse tema esimene element, milleks on üks sõne, kordamiseks ekraanile ja kutsutakse välja `kordaTähed` sel sõnel. Spetsifikatsiooni põhjal edastab see protseduur tõeväärtuse, mis näitab kordamise õnnestumist; see tõeväärtus satub muutujasse `b`. See, et True korral jätkataks järgmise reaga, False korral aga sama reaga, on saavutatud nii, et True korral jäetakse rekursiivsel pöördumisel listist esimene element ära, kuid False korral toimub rekursiivne pöördumine sama listiga uuesti.

Tühja argumentlisti korral pole vaja midagi teha, sest lõputeate kirjutab põhiprogramm. Seepärast on teise deklaratsiooni paremaks pooleks lihtsalt `return ()`.

Viimaks muutuja `kordaTähed` definitsiooniks võiks olla

```
kordaTähed (c : cs)
  = do
      d <- getChar
      if d == c then kordaTähed cs else return False
kordaTähed _
  = return True
```

(84)

Kui kasutaja sisestab õige tähe, siis jääb tal veel ülejäänud tähed õigesti korrata, seepärast on selle juhu jaoks programmeeritud rekursiivne pöördumine listi sabal. Kui kasutaja sisestab vale tähe, siis on kordamine ebaõnnestunud, seetõttu sel juhul

muud ei pea tegema kui edastama tõeväärtuse False. Tühja listi juht tuleb käiku ainult siis, kui kasutaja on kogu rea õigesti sisestanud, seepärast võib ka siin lõpetada töö, edastades tõeväärtuse True.

Pangem tähele, et rekursiivse pöördumise juhutudel koodis (83) ja (84) operaator `return` puudub. Üldtuntud imperatiivsetes keeltes tuleb tagastusoperaator ka sellisesse kohta märkida, kuid Haskellis oleks `return` siin vale. Et Haskelli `return` tähendab muud — ta teeb andmest protseduuri, mis seda annet edastab — ja `kordaTähed cs` väärtus juba on protseduur, annaks `return` lisamine tüübivea.

Ülesandeid

211. Defineerida muutuja `grammar` väärtuseks karritatud funktsioon, mis võtab argumendiks täisarvu n ja sõne s ja annab tulemuseks protseduuri, mis kirjutab ekraanile n korda sõnet s koos reavahetusega.

212. Defineerida muutuja `loeList` väärtuseks funktsioon, mis võtab argumendiks listi ja annab väärtuseks protseduuri, mis tsüklis ootab kasutajalt klahvivajutust: kui see on *enter*, kirjutab ekraanile listi järgmise elemendi; kui see on *escape*, lõpetab; kui see on midagi muud, ei reageeri kuidagi. Kui list on läbi, lõpetab programm kohe töö. Ekraanipilt peab olema loetav ja ilma ülearuste tühjade ridadeta.

    Muutujat `loeList` kasutades käivitada interaktiivses keskkonnas protseduur, mis kirjutab kasutaja *enter*-vajutuste peale koodide järjekorras kõik sümbolid. Sama muutujat kasutades käivitada ka protseduur, mis kirjutab kasutaja *enter*-vajutuste peale suuruse järjekorras naturaalarvude ruudud.

213. Kirjutada muutuja `harjutus` definitsioon (81) ümber nii, et tippimist harjutav programm võtaks teksti reakaupa failist, mille nime ta töö algul küsiks.

214. Kirjutada klaviatuuril tippimist harjutav programm ümber nii, et vea korral programm rida uuesti tippida ei laseks, kuid lõpus väljastaks ekraanile info, mitu rida mitmest õigesti tipiti.

2. Otseselt rekursiivsed protseduuridefinitsioonid. Oleme harjunud, et argument rekursiivsel väljakutsel peab mingis mõttes vähenema. Protseduuride juures võib selle reegli rahuga unustada. Interaktiivsetes protsessides sõltub lõpetamine või jätkamine protsessi kulust ja võib tuua palju näiteid ka protseduuridest, mis jäävad lõpmatusse tsüklisse, nii et nende töö on seejuures mõttekas.

Muutuja `kordaRead` definitsioonis (83) esineb rekursiivne pöördumine samal argumendil mis sisse tuli. See jätab õhku võimaluse, et protseduur ei lõpetagi, kuid

tema spetsifikatsioonist tulenevalt on see loomulik: kui kasutaja ikka jääbki sama rea juures vigu tegema, siis programm peab jäämagi temaga seda rida harjutama.

Teame, et kui rekursiivse pöördumise argument on igal pool sama mis sisse tuleb, siis saab selle argumendi kaotada.

Sellise optimeeringu tegime muutuja `kaheksLahuta` juures, kus definitsioonist (73) sai (74).

Kui õnnestub kõik argumendid rekursioonist välja pukseerida, siis see, mis jääb rekursiivselt defineerituks, pole enam funktsioonitüüpi. Nii võib jõuda protseduuri otseselt rekursiivse kirjelduseni.

Olgu meil näiteks vaja protseduuri, mis jälgib standardsisendist tulevaid sümboleid ja lõpetab ainult siis, kui tuleb teatav kindel sümbol. Selleks kodeerime funktsiooni, mis võtab argumendiks selle sümboli, mille peale töö lõppema peab, ja annab tulemuseks vastava protseduuri, vastavana signatuurile

```
kuulaKuni
  :: Char -> IO ()
```

Seda protseduuri võib kasutada olukorras, kus tarvis oodata kasutajalt kinnitust, et võib tööga edasi minna. Näiteks on mõttekas väljakutse koodist

```
jätkab
  :: Char -> IO ()
jätkab a
  = do
      putStrLn ("Jätkab sümbol " ++ show a ++ ".") ,
      hSetBuffering stdin NoBuffering
      hSetEcho stdout False
      kuulaKuni a
      hSetEcho stdout True
```

kui põhiprogramm on näiteks

```
main
  = do
      putStr "Ruudud: "
      print [x ^ 2 | x <- [0 .. 255]] .
      jätkab 'X'
      putStr "Kuubid: "
      print [x ^ 3 | x <- [0 .. 255]]
```

Avaldise `jätkab 'X'` väärtuseks on protseduur, mis ootab tummalt standardsisendist sümboleid kuni suure X-täheni, siis lõpetab töö.

Esmane versioon `kuulaKuni` definitsioonist võib olla

```
kuulaKuni a
  = do
      c <- getChar
      if c == a then putChar '\n' else kuulaKuni a
```
. (85)

Paneme tähele, et koodis (85) on rekursiivne pöördumine muutuja `kuulaKuni` poole muutmata argumendiga. Lihtsustumine puudub täiesti, arvutusprotsessi lõpetamine sõltub siin muudest asjaoludest.

Viies rekursiooni ilma argumendita `a` lokaalsesse definitsiooni, on tulemuseks kood

```
kuulaKuni a
  = let
      proc
        = do
            c <- getChar
            if c == a then putChar '\n' else proc
    in
    proc
```
.

Siin rekursiivselt defineeritava lokaalse muutuja `proc` väärtus on protseduur.

Samamoodi otsese rekursiooniga on kirjeldatav näiteks protseduur, mis kuulab standardsisendit, kuni sisestatakse tühi rida, leiab sisestatud ridade arvu ning edastab selle: sobivad signatuur ja definitsioon

```
loendaRead
  :: (Integral a),
  => IO a

loendaRead
  = do
      rida <- getLine
      if null rida
        then return 0
        else do
               ridadeArv <- loendaRead
               return (1 + ridadeArv)
```
. (86)

Ülesandeid

215. Defineerida muutuja `pööra` väärtuseks protseduur, mis tsüklis küsib eesti keeles kasutajalt sisendit, loeb klaviatuurilt rea ja toob rea lõppedes ekraanile selle rea sümbolid vastupidises järjekorras, kuni sisestatakse tühi rida.

216. Kasutades ülesandes 177 defineeritud muutujat `niheEdastusel`, anda muutuja `püsivNihe` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab tulemuseks protseduuri, mis kuulab standardsisendit kuni reavahetuseni ja jooksvalt kajastab loetud sümbolid ekraanil, seejuures tähed tsüklilises ladina tähestikus n-kohalise nihkega. Rekursioon viia protseduuritasemele.

217. Kirjutada programm koodiga (86) antud muutuja `loendaRead` testimiseks.

218. Defineerida muutuja `loendaSümbolid` väärtuseks protseduur, mis kuulab standardsisendit kuni tühja reani ja edastab loetud ridades olevate sümbolite koguarvu (ilma reavahetussümboliteta). Kirjutada testprogramm.

4.1.3 Rekursiivselt defineeritud andmestruktuurid

Nagu oleme kogenud, võib Haskellis täiesti korrektse ja mõtteka avaldise väärtuseks olla lõpmatu andmestruktuur, näiteks list. Seni oleme lõpmatuid liste tekitanud vaid aritmeetilise jada erisüntaksi abil, lõpmatuid liste välja andvaid eeldefineeritud funktsioone kasutades ja selliselt genereeritud lõpmatuid liste modifitseerides.

Kui tahetakse defineerida omal soovil mõni täiesti uus lõpmatu list, siis ei saa üle ega ümber rekursioonist, kuna lõpmatu listi tekitamiseks on vaja programmeerida lõpmatu arvutusprotsess, mis lõpliku eeskirjaga kirjeldatuna peab sisaldama tsüklit ja mida funktsionaalses keeles järelikult ilma rekursioonita väljendada ei ole võimalik.

1. Definitsioonid läbi funktsiooni. Siingi on üks võimalus sihile jõuda funktsioonide kaudu, väärtuseks siis lõpmatud listid.

Proovime kirjeldada funktsiooni, mis annab listina välja geomeetrilise jada, argumentideks esimene element ja tegur. Signatuur olgu

```
geom
  :: (Num a)      .
  => a -> a -> [a]
```

Tulemuslisti esimese elemendiga raskusi ei ole, sest see tuleb otse argumendist. Listi ülejäänud osa (saba) võib kirjeldada mitme ideega.

Üks variant on panna tähele, et geomeetrilise jada liikmed alates teisest moodustavad geomeetrilise jada sama teguriga. Et teine liige ise on esimese liikme ja teguri korrutis, saame definitsiooni

```
geom a q
  = a : geom (a * q) q.
```
(87)

Aga on ka võimalik võtta aluseks sama tähelepanek interpreteerituna nii, et tulemuslisti saba iga element peab olema kogu listi sama järjekorranumbriga elemendi ja jada teguri korrutis (sest saba element on kogu listi sama järjekorranumbriga elemendist jadas järgmine). Selle idee põhjal saame koodi

```
geom a q
  = a : [x * q | x <- geom a q].
```
(88)

See kood töötab, sest tänu laisale väärtustamisele saab rekursiivse pöördumise tulemuseks olevat listi otsast kasutama hakata juba enne, kui ta lõpuni valmis on (tegu on lõpmatu listiga, nii et ta lõpuni valmis ei saagi). Täpsemalt, vastavalt definitsioonile oskab `geom a q` esimese elemendi paika panna; seega ka rekursiivne pöördumine annab ühe elemendi, mis kohe haaratakse korrutisse `q`-ga ja saadakse kogu listi teine element. Seega annab ka rekursiivne pöördumine teise elemendi, mis jällegi kohe korrutisse haaratakse ja saadakse kogu listi kolmas element jne.

Kuid arvutus selle definitsiooni järgi on tunduvalt aeglasem kui algse definitsiooni (87) järgi. Lihtne on mõista, miks. Definitsiooni (87) korral paneb operaatori väljakutse paika listi elemendi `a`, valmistab ette korrutise `a * q` ja kõik edasine on juba rekursiivse pöördumise teha. Igal rekursioonitasemel tehakse ühepalju tööd, mistõttu töömaht on võrdeline valmisehitatud listijupi pikkusega. Seevastu definitsiooni (88) korral tuleb korrutada rekursiivse pöördumise tulemuse iga elementi. Et sama toimub ka teistel rekursioonisammudel, on töömaht kokkuvõttes võrdeline valmisehitatud listijupi pikkuse ruuduga.

Teisiti öeldes, definitsiooni (88) ebaefektiivsus tuleb sama asja korduvast arvutamisest. Listi iga element arvutatakse algusest peale eraldi välja, samas kui definitsiooni (87) puhul leitakse iga järgmine element jooksva elemendi kaudu.

2. Otseselt rekursiivsed listidefinitsioonid.

2.1. Muutujate väljaviimine rekursioonist. Kaotades rekursioonist muutumatuid argumente, on võimalik jõuda ka listi otseselt rekursiivse definitsioonini. Tänu laisale väärtustamisele töötavad Haskellis ka sellised.

See, et arvutus definitsiooni (88) järgi on mõttetult aeglane, ei tähenda, et tema aluseks olnud idee on halb. Kuna argumendid jäävad rekursiivsete pöördumiste käigus

samaks, võime tõsta nad mõlemad rekursioonist välja ja saada definitsiooni

```
geom a q
  = let
      gs
        = a : [x * q | x <- gs]
    in
    gs
```
(89)

Lokaalne deklaratsioon defineerib rekursiivselt muutuja `gs`, mille väärtus on list.

Ehkki arvutus definitsiooni (89) järgi toimub põhimõtteliselt samamoodi nagu koodi (88) puhul, käib see kummatigi niisama kiiresti kui koodi (87) puhul. Paljas argumentide kaotamine rekursioonist on viinud olulise võiduni tööajas.

Nimelt on arvutusprotsess korraldatud nii, et muutujaid väärtustatakse samas tähenduses alati ainult üks kord. Arvutusprotsessi iga lõim, mis muutujat kasutab, jätab ta teiste jaoks maha kujul, kuhu ta tema väärtustab. Sama etappi muutuja väärtustamisest pole enam vaja korrata.

Kui funktsioonitüüpi muutuja rakendamisel erinevatele argumentidele toimub iga kord uus arvutus (sest tulemus ju sõltub argumendist, tegu pole iga kord sama muutuja väärtusega), siis definitsiooni (89) korral arvutatakse otse muutujat `zs` (argumenti pole), arvutatakse ainult üks list. Seetõttu toimub tegelikult ainult üks rekursiivne pöördumine, sellest edasi ei ole definitsiooni rekursiivsusel enam mingit tähtsust. Kõik käib samamoodi, nagu arvutataks listi elemente mõne teise, juba olemasoleva listi põhjal, kuigi tegelikult need listid juhtumisi langevad kokku. See töötab, sest kirjutav lõim on lugevast ühe elemendi võrra ees. Arvutuse maht kasvab jälle lineaarselt valmisehitatud algusjupi pikkuse suhtes.

2.2. Üldised printsiibid. Võib märkida, et andmestruktuuri rekursiivse defineerimise printsiibid on mõnevõrra teistsugused kui funktsioonide puhul, mis struktuuri järgi midagi välja arvutavad. Viimastele omane baasjuht pole vajalik, sellega seoses langeb ära ka hargnemise vajadus.

Rekursiooni baasi rollis on nõue, et kuni arvutus pole lõppenud, peab kogu aeg olema võimalik lõpliku ajaga mingi andmestruktuuri tükk juurde ehitada. Muidu jääb protsess tühja tsüklisse ja mingit struktuuri ei teki. Listi puhul tähendab see nõue, et definitsioonis tuleb vähemalt üks element listi algusest kirjeldada ilma rekursioonita.

Rekursiooni samm aga ütleb, kuidas andmestruktuuri ülejäänud osa arvutada, kasutades selleks sama eeskirja või loodavat struktuuri ennast.

Veidi filosoofiliseks minnes võib öelda, et erinevus struktuure tarbivast algoritmist on selles, et rekursiooni baas ja samm orienteeruvad mitte argumendile, vaid tulemusele. Baas ja samm siin on mitte juhud, vaid etapid.

Lihtsaimad definitsioonid, kus list esitatakse iseenda kaudu ilma funktsiooni vahenduseta, on võrdsete elementidega lõpmatutel listidel. Näiteks lõpmatul listil, mille iga element on 0, võrdub pea 0-ga ja saba listi endaga. Siit saame deklaratsiooni

```
zs
  = 0 : zs.
```
(90)

Vaatame ka keerulisemaid näiteid. Oletame, et tahame arvutada listi, mille elementideks oleksid kasvavas järjekorras kõik positiivsed täisarvud, mille esituses algarvude astmete korrutisena ehk kanoonilises esituses ei esine muid algarve peale 2 ja 5. Teisi sõnu, need arvud tohivad algarvudest jaguda vaid 2- ja 5-ga. Signatuur oleks

```
kords
  :: (Integral a).
  => [a]
```

Esimene liige selles listis on kindlasti 1, sest 1 on vähim positiivne täisarv ja keelatud algarvudega ta ei jagu. Iga ülejäänud arv n selles listis on mingi positiivse täisarvu n' 2- või 5-kordne, kusjuures ka n' ei jagu muude algarvudega peale 2 ja 5 ning ta on n-st väiksem. Seega iga arv meie listi sabas on mingi listis temast eespool esineva arvu 2- või 5-kordne. Samas kui meie listi kõiki elemente korrutada 2-ga ja 5-ga, siis tekivad meie listi elemendid; arvestades eelnevat vaatlust, saame sellisel viisil niisiis parajasti kõik listi saba elemendid.

Arutlusest tuleneb, et otsitava listi saba arvutamiseks terve listi kaudu tuleks leida selle listi elementide 2-kordsete list ja 5-kordsete list, seejärel aga panna nende kahe listi elemendid kasvavas järjestuses ühte kokku, kaotades ka kordused. Kuna liste hoitakse kogu aeg elementide kasvavas järjestuses ja 2- või 5-ga korrutamine seda omadust ei muuda, siis on vaja operatsiooni, mis kahe kasvavas järjestuses elementidega listist paneks kokku ühe kasvavas järjestuses elementidega listi. Sellist operatsiooni nimetatakse põimimiseks. Selle saame koodiga

```
põim xs@ (a : as) ys@ (b : bs)
  = case compare a b of
      LT
        -> a : põim as ys
      GT
        -> b : põim xs bs
      _
        -> a : põim as bs
põim xs           []
  = xs
põim _            ys
  = ys
```
(91)

Kuna see operatsioon kasutab andmete järjestusvahekordi, on ta kasutatav järjestusega tüüpi elementidega listidel ehk kõige üldisem signatuur on

```
põim
  :: (Ord a)          .
  => [a] -> [a] -> [a]
```

Nüüd võime muutuja `kords` defineerida koodiga

```
kords
  = 1 : põim [x * 2 | x <- kords] [x * 5 | x <- kords]
```

Defineerime sama tehnikaga muutuja `fibs` väärtuseks listi, mis sisaldab parajasti kogu Fibonacci jada. Sobivad signatuur ja definitsioon

```
fibs
  :: (Integral a),
  => [a]

fibs
  = 0 : 1 : kahesummad fibs'
```
(92)

kus muutuja `kahesummad` on varem antud definitsiooniga (75).

Definitsiooni (92) järgi toimub Fibonacci arvude arvutamine lineaarse ajaga järjekorranumbri suhtes, iga järgneva elemendi leidmiseks tehakse vaid üks liitmine. Definitsiooni (68) kõrvale võib seega tuua veel ühe n-ndat Fibonacci arvu lineaarse ajakuluga arvutamist realiseeriva definitsiooni

```
fib n
  | n >= 0
    = fibs !! n                              ,
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
(93)

mille ideeks on valmisarvutatud listist lugemine.

Ülesandeid

219. Otsida Hugsi teegist üles muutujate `repeat` ja `cycle` definitsioonid ja saada neist aru.

220. Defineerida muutuja `sq2s` väärtuseks list, mille elemendid oleksid järjekorras tabeli (69) veerud paarikujul.

221. Defineerida muutuja `rotatsioonid` väärtuseks funktsioon, mis võtab argumendiks listi ja kui see on lõplik, siis leiab tema kõik tsüklilised nihked

(list ise, siis nihe ühe koha võrra, kus otsast välja nihkuv element tuleb teisest otsast sisse, jne) listina, mille pikkus võrdub argumentlisti pikkusega.

222. Defineerida muutuja `hamming` väärtuseks list, mille elementideks on kasvavas järjekorras kõik positiivsed täisarvud, mille kanoonilises esituses ei esine muud algarvud peale 2, 3, 5.

223. Ülesandes 186 defineeriti Lucas' jada. Defineerida muutuja `lucs` väärtuseks list, mille elementideks on järjest kõik Lucas' jada liikmed.

224. Anda muutujale `tuia` väärtuseks list elementidega $1, 2, 3, 2, 3, 4, 3, 4, 5, \ldots$.

225. Defineerida muutuja `venivillem` väärtuseks funktsioon, mis võtab argumendiks listi l ja annab väärtuseks listi, mis algab listi l elementidega, millele järgnevad listi l elemendid igaüks kahekordselt, seejärel järgnevad listi l elemendid igaüks neljakordselt, edasi kaheksakordselt jne.

226. Defineerida muutuja `bitisõned` väärtuseks list, mille elementideks on kõik lõplikud numbritest 0 ja 1 koosnevad sõned, igaüks üks kord.

227. Defineerida muutuja `kaheAstendajad` väärtuseks list, milles on järjest arvu 2 astendajad kõigi positiivsete täisarvude kanoonilises esituses.

2.3. Lõpliku listi defineerimine. Lõpmatu andmestruktuuri rekursiivne definitsioon, olgu siis läbi funktsiooni või otsene, on lõpliku omast selles mõttes lihtsamgi, et lõplikul juhul tuleb tegelda ka lõpetamistingimustega. Kuid ka lõplikke liste saab teinekord kirjeldada otsese rekursiooniga. Definitsioonist ei pruugi lõplikkus/lõpmatus üldse silmaga näha olla.

Vaatame näitena Collatzi jadade leidmist. Kui n on positiivne täisarv, siis vastavaks Collatzi jadaks nimetatakse järgmiste reeglite alusel arvutatud arvujada:

- esimene liige on n;
- kui senileitud liikmetest viimane on 1, siis sellega on jada lõppenud;
- kui senileitud liikmetest viimane on paaris, siis järgmine liige on täpselt pool sellest;
- kui senileitud liikmetest viimane on paaritu, siis järgmine liige on ühe võrra suurem kui selle liikme kolmekordne.

Kodeerime Collatzi jada arvutusreeglid Haskellis, andes rekursiivse definitsiooni funktsioonile, mille väärtus positiivsel argumendil n on n-ga algav Collatzi jada;

signatuuriks sobib

```
clz
  :: (Integral a).
  => a -> [a]
```

Paneme tähele, et kui $n > 1$, siis Collatzi jada esimese elemendi väljajätmisel saame samuti Collatzi jada, nimelt selle, mis vastab algse jada teisele elemendile. Nii saame koodi

```
clz n
  | n > 0
    = n :
      if n == 1
        then []
        else
          let
            (q , r)
              = n `divMod` 2
          in
          clz (if r == 0 then q else n * 3 + 1)
  | otherwise
    = []
```
. (94)

Kuid Collatzi jadu saab defineerida ka otsese rekursiooniga, nagu näitab kood

```
clz n
  | n > 0
    = let
        f (x : xs)
          | x == 1
            = []
          | otherwise
            = let
                (q , r)
                  = n `divMod` 2
              in
              (if r == 0 then q else x * 3 + 1) :
              f xs
        cs
          = n : f cs
      in
      cs
  | otherwise
    = []
```
.

Ideeks on kirjeldada abifunktsioon (koodis muutuja `f` väärtus), mis arvutatava listi järgi tema saba annab. Lokaalset muutujat `f` rakendatakse arvutuse käigus ainult argumentidele, mille väärtus on mittetühi list, mistõttu pole vajadust tühja argumentlisti juhu lisamisele tema definitsiooni.

Kumbki definitsioon ei näita välja, kas Collatzi jada on lõplik või lõpmatu. Interaktiivses keskkonnas võib erinevaid Collatzi jadu välja arvutada ja näha, et nad kipuvad lõppema. Tegelikult ei ole üldse teadagi, kas leidub selline n, mille korral Collatzi jada on lõpmatu. See on üks kuulsaid matemaatika lahendamata probleeme.

Ülesandeid

228. Defineerida muutuja `kahendPööratud` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see on mittenegatiivne, siis annab tulemuseks listina arvu n kahendesituse numbrid ilma algusnullideta tavalisele vastupidises (järgu kasvamise) järjekorras. Negatiivsel argumendil peab arvutus lõppema adekvaatse veateatega.

229. Defineerida muutuja `kümnendmurd` väärtuseks funktsioon, mis võtab argumendiks ratsionaalarvu q ja annab väärtuseks paari, mille esimene komponent on q täisosa ja teine komponent on list q murdosa kümnendesituse numbritest nende esinemise järjekorras (lõpliku esituse puhul peab list olema lõplik, lõpmatu esituse puhul lõpmatu). Soovitav on kasutada ülesandes 154 või 157 kirjutatud muutujat `õigeProperFraction`.

230. Olgu $\wr$ binaarne operatsioon, mis defineeritakse reegliga

$$a \wr r = a + \frac{1}{r}.$$

(Sama operatsioon defineeriti muutuja `//` väärtuseks koodiga (24); sarnaselt sellega kasutame siin tähist $\wr$ paremassotsiatiivselt.)

Reaalarvu r **ahelmurruks** nimetatakse lõplikku või lõpmatut esitust vastavalt kujul $r = a_0 \wr \ldots \wr a_n$ või $r = a_0 \wr a_1 \wr \ldots$, kus kõik a_i on täisarvud ja ainult a_0 võib olla mittepositiivne, kusjuures lõplikkuse korral $a_n > 1$. Arve a_i nimetatakse arvu r **osajagatisteks**.

Defineerida muutuja `ahel` väärtuseks funktsioon, mis võtab argumendiks ratsionaalarvu q ja annab välja q osajagatiste listi. Soovitav on kasutada ülesandes 154 või 157 defineeritud muutujat `õigeProperFraction`.

3. Muud rekursiivsed definitsioonid. Rekursiivselt võib Haskellis määratleda põhimõtteliselt mida iganes.

Näiteks võib koodiga (70) defineeritud operaatori `järgmVeerg` samaväärselt defineerida koodiga

```
järgmVeerg (u , v)
  = let
      p = (u + v , u + fst p) ,
    in
    p
```

kus lokaalse deklaratsiooniga kirjeldatakse rekursiivselt paar. Sama asja võib ümber kirjutada ka kujul

```
järgmVeerg (u , v)
  = let
      p@ (s , _)
        = (u + v , u + s) ’
    in
    p
```
(95)

kus rekursiivne ainult muutuja `s`, mis tähistab paari vasakut komponenti.

Operaatori `järgmVeerg` defineerimine rekursiooniga on muidugi ainult naljanumber, sest seal pole tsüklit, vaid konstantne arv samme.

Rekursioon võib Haskellis võtta lausa paradoksaalseid vorme, mis astuvad vastu meie tavaintuitsioonile sellest, kuidas on võimalik asju defineerida.

Olgu meil vaja anda muutuja `üleÜheEtte` väärtuseks funktsioon, mis võtab argumendiks listi l ja annab tulemuseks listi, mille alguses on listi l elemendid paarisarvulise indeksiga kohtadelt ja seejärel l ülejäänud elemendid. Signatuur on

```
üleÜheEtte
  :: [a] -> [a]
```

Esmane võimalus seda teha on kasutada definitsiooniga (76) või (77) või (80) antud muutujat `kaheksLoe`. Tema rakendamine argumentlistile teeb jaotamistöö ära ja jääb üle vaid paari komponendid ühte listi kokku konkateneerida. Selle ideega saame koodi

```
üleÜheEtte xs
  = let
      (us , vs)
        = kaheksLoe xs
    in
    us ++ vs
```

Selle lahenduse puudus on, et `kaheksLoe` rakendamisel koostatava paari esimene komponentlist konkateneerimise käigus kopeeritakse. Tühja töö vältimiseks tuleks `kaheksLoe` definitsioon asendada sellisega, mis ehitab paari esimese listi kohe õigesse kohta — teise listi ette.

Võib tunduda, et funktsionaalse paradigma raames pole võimalik seda saavutada, sest mäluhaldus on kirjeldamatu. Kuid siiski — Haskelli avaldiste väärtustamine on laisk ja seal töötavad rekursiooniskeemid, mis agaras keeles on kujuteldamatud. Kirjutame `kaheksLoe` definitsiooni, mis kirjeldab rekursiooni baasjuhus väljaantava väärtuse selle listi kaudu, mis saadakse teise komponendina paarist, mis funktsiooni rakendamisel algsele argumendile tulemuseks on. Võttes aluseks `kaheksLoe` definitsiooni (76), saame niimoodi `üleÜheEtte` definitsiooni

```
üleÜheEtte xs
  = let
      kaheksLoe (x : y : ys)
        = let
            (us , vs)
              = kaheksLoe ys
          in
          (x : us , y : vs)   ,
      kaheksLoe xs
        = (xs ++ vs , [])
      (us , vs)
        = kaheksLoe xs
    in
    us
```
(96)

mis tõepoolest ka töötab.

Koodis (96) defineeritakse lokaalselt paari `(us , vs)` väärtuseks operaatori `kaheksLoe` töö tulemus argumendil, millega `üleÜheEtte` välja kutsuti, kusjuures `kaheksLoe` definitsioon oma baasjuhus pöördub sellesama paari teise komponendi poole. Nii on `kaheksLoe` ja `vs` definitsioonid vastastikku rekursiivsed.

Baasjuht kasutab kogu rekursiivse arvutuse tulemust, mis ju pole baasjuhu töötlemise ajal veel teada! Haskelli laiskuse tõttu aga väärtustaja seda ei "märka" ja viga ei tule. Edasise väärtustamise käigus selgub ka `vs` ehitus ja kokkuvõttes kujuneb lõpptulemuseks soovitud paar.

Analoogiline lahendus ei ole võimalik, kui `kaheksLoe` defineerimisel võtta aluseks definitsioon (77), sest baasjuhu juures pole teada, kumba komponentidest kujuneb esimene, kumba teine list, ja nii ei ole võimalik üheselt otsustada, kumb komponent peab sisaldama muutujat `vs`.

Demonstreerime sama lahendusvõtet veel ühe näite peal. Olgu meil vaja saada muutuja `sumSees` väärtuseks predikaat, mis võtab argumendiks arvulisti ja kontrollib,

kas tema elementide summa on ise selle listi element. Signatuur oleks

```
sumSees
  :: (Num a)     .
  => [a] -> Bool
```

Moodulist Prelude saame operaatori `elem`, mille väärtuseks on funktsioon, mis võtab argumendiks andme x ja sama tüüpi elementidega listi l: kui x esineb listis l, annab ta tulemuseks True, vastasel korral kui l on lõplik, siis False. Seega olemasolevate operaatorite kaudu tehes on lahenduseks kood

```
sumSees xs
  = elem (sum xs) xs .
```

Selle definitsiooni järgi arvutamisel läbitakse argumentlist kaks korda: üks kord operaatori `sum` rakendamisel ja teine kord operaatori `elem` rakendamisel.

On selge, et selle kohta, kas listi elementide summa listis esineb, ei ole võimalik saada vähimatki informatsiooni enne, kui summa on välja arvutatud ja seega list lõpuni käidud. Pärast summa selgumist, kui juhtumisi näiteks summa listis ei esine, on selle kindlakstegemiseks vaja veelkord kõiki listi elemente külastada.

Võtame aga kasutusele lokaalse abioperaatori `sumSees`, mille ülesandeks on arvutada otsitav tõeväärtus paaris listi elementide summa endaga. Selle operaatori definitsioonis tähistagu `(b , s)` tema rekursiivse pöördumise tulemust listi sabal. Lokaalse operaatori `sumSees` algsele argumentlistile rakendamise tulemuse seome väljaspool tema definitsiooni paariga `(bx , sx)`. Siis saab lokaalse operaatori definitsioonis lõppsummat `sx` kasutada. Tekib kood

```
sumSees xs
  = let
      sumSees (z : zs)
        = let
            (b , s)
              = sumSees zs
          in
          (z == sx || b , z + s) ’
      sumSees _
        = (False , 0)
      (bx , sx)
        = sumSees xs
    in
    bx
```
(97)

mis tõesti ka töötab.

Seega ülesanne, kus on vaja listi elemendid kaks korda läbi vaadata, seejuures teine kord peale esimese korra lõppu, on realiseeritud listi ühekordse lugemisega.

Saamaks aru, kuidas see paradoks on võimalik, on vaja täpselt mõista, mida tähendab listi lugemine. Selle väljendiga märgitud tegevus, mis koodi (97) puhul toimub ühekordselt, on täpsemalt öeldes listi struktuuri avamine. Listi elemente aga loetakse kaks korda.

Võrdluseks, ka tavalistel juhtudel nagu näiteks `suurSumma` definitsioon (63) tekitab rekursioonisügavustesse minek mällu vaid pika väärtustamata avaldise, reaalne arvutus toimub alles sealt tagasipöördumisel. See on nii isegi sõltumata laisast väärtustamisest, sest positiivse argumendi juht definitsioonis (63) sätestab tulemusena summa, mille esimene liidetav on rekursiivse pöördumise tulemus — järelikult liitmine pole võimalik enne, kui rekursiivne pöördumine on töö lõpetanud.

Sama lugu on muidugi ka definitsiooniga (97), vaid tekkiv avaldis on keerulisem. Aga kui niikuinii koostatakse ainult väärtustamata avaldist, siis polegi oluline, kas tema see või teine osa on hetkel teada olevate andmete järgi reaalselt arvutatav või mitte. Proovima asutakse alles siis, kui ollakse jõudnud baasjuhuni. Kuid siis on kõik listi elemendid seal pikas väärtustamata avaldises omal kohal sees. Seepärast ei olegi mingit tarvidust neid enam listist lugema hakata. Kõik toimub täiesti loomulikult: väärtustatakse kõigepealt see osa avaldisest, mis puudutab elementide summat, sest seda on tõeväärtuse leidmiseks vaja, ning seejärel tõeväärtus.

Ülesandeid

231. Näidata definitsioonis (96) nimede `xs`, `us` ja `vs` kõigi esinemiste kohta nende tähendusulatus (skoop).

232. Kirjutada definitsioon (96) samaväärselt ümber ilma paarinäidisteta, kasutades muutujaid `fst` ja `snd`. Millised muutujad on nüüd defineeritud vastastikrekursiivselt ja mis on nende väärtus?

233. Kirjutada muutujale `üleÜheEtte` definitsiooni (96) eeskujul alternatiivne definitsioon, kus lokaalse operaatori `kaheksLoe` defineerimisel oleks eeskujuks võetud definitsioon (80).

234. Anda muutuja `negEtte` väärtuseks funktsioon, mis võtab sisse reaalarvude listi l ja annab välja listi, mille alguses on l negatiivsed elemendid ja seejärel l mittenegatiivsed elemendid. Vältida tühja tööd arvutusel.

235. Defineerida muutuja `kolmeJärgi` väärtuseks funktsioon, mis võtab argumendiks täisarvude listi l ja annab välja listi, mille alguses on l elemendid, mis annavad 3-ga jagades jäägi 0, nende järel elemendid, mis annavad jäägi 1, ning siis elemendid, mis annavad jäägi 2. Vältida tühja tööd arvutusel.

236. Defineerida muutuja `keskmista` väärtuseks funktsioon, mis võtab argumendiks ujukomaarvude listi ja annab tulemuseks tema igast elemendist kõigi elementide aritmeetilise keskmise lahutamisel saadava listi. Arvutamisel peab listi loetama ainult üks kord.

237. Defineerida muutuja `ilmaÜhetaSummad` väärtuseks funktsioon, mis võtab argumendiks mingi arvulisti l ja annab välja listi, milles on samapalju elemente kui l-s ja mille element kohal i võrdub summaga listi l kõigist elementidest peale selle, mis on kohal i. Arvutamisel peab listi loetama ainult üks kord.

238. Defineerida muutuja, mille väärtustamisel tekib lõpmata palju veateateid.

4.2 Arvutused formaalsete parameetrite peal

Imperatiivses keeles programmeerides hoitakse arvutuse jaoks pidevalt vajaminevad andmed tüüpiliselt muutujates ja vajadusel uuendatakse neid omistuskäskudega. Funktsionaalses keeles pole muutujaid sellises mõttes nagu on imperatiivses keeles, omistus puudub. Sellegipoolest on funktsionaalses keeles võimalik arvutamine samasugusel põhimõttel, kasutades vahetulemuste hoidmiseks rekursiivselt defineeritud operaatori argumente.

Üksteisele järgnevate rekursiivsete pöördumiste käigus sellised argumendid ei lihtsustu, tüüpiliselt on hoopis vastupidi. Tavaliselt ei ole nende roll näidata, millal rekursiivsete pöördumiste jada lõpetada. Kui vaja, tehakse see otsus teiste argumentide või mõne spetsiaalse lõpetamistingimuse põhjal.

Taolised argumendid on oma iseloomult samasugused nagu need, mis tulid ette lõpmatu listi defineerimisel. Seega oleme sellistega juba tuttavad.

Kuid senistes näidetes tulenesid kõik argumendid loomuldasa ülesande püstitusest.

Sageli selliselt arvutamiseks ülesande püstitusest otse tulenevatest argumentidest ei piisa, vaid on vaja rekursioon viia abioperaatorisse, millel on spetsiaalsed lisaargumendid. Seetõttu see lähenemine tihti vastandub otsesele rekursioonile.

Funktsionaalses keeles näeb imperatiivset paradigmat ahviv programm muidugi välja üsna teistmoodi kui imperatiivses keeles. Taoline stiil koodi loetavust vaevalt et parandab. Seepärast kui muud eesmärki lisaargumentidega arvutamisel pole, siis on parem kirjutada kood otsese rekursiooniga või kombineerides juba defineeritud operaatoreid.

4.2.1 Akumulaatorid

Akumulaator on rekursiivselt defineeritud operaatori argument, mis arvutuse käigus kogub endasse infot, akumuleerib seda. See tähendab, et rekursiivsel pöördumisel lisatakse sinna argumenti midagi juurde võrreldes tema varasema väärtusega. Akumuleeritav info võib olla väga mitmesugune.

1. Loendurid. Lihtsamal juhul on akumulaatori ülesandeks millegi loendamine; nimetame sellist parameetrit loenduriks.

Näiteks oli koodiga (72) defineeritud muutuja `nullideArv` väärtuseks funktsioon, mis arvulisti järgi annab välja tema nulliga võrduvate elementide arvu. Kood (72) on otseselt rekursiivne. Kuid sama eesmärgini võib jõuda ka loenduriga.

Võtame kasutusele lokaalse operaatori `arv`, millel lisaks listiargumendile on üks arvuline argument, loendur `n`. Algsel väljakutsel anname listiargumendiks põhioperaatori argumendi, loenduriks aga nulli. Rekursiivsel pöördumisel anname loenduri edasi kas 1 võrra suurendatuna, kui listist loeti 0, või muutmata, kui listist loeti midagi muud. Listiargumendiks anname jooksva listi saba, sest pea on ära töödeldud.

Selle kava järgi arvutades on lisaargumendi väärtus ilmselt kogu aeg võrdne senileitud nullide arvuga, st ta toimib loendurina, ja kui list läbi saab, on seal parajasti nullide arv kogu listis. Seetõttu võib töö lõpus lihtsalt selle argumendi välja anda. Definitsioon, mis selle algoritmi realiseerib, on

```
nullideArv xs
  = let
      arv n (z : zs)
        = arv (if z == 0 then n + 1 else n) zs
      arv n _
        = n
    in
    arv 0 xs
```
. (98)

Ülesandeid

239. Defineerida muutuja `peaEsinemisteArv` väärtuseks funktsioon, mis argumentlisti järgi leiab arvu, mitu korda esineb listi pea selles listis.

240. Anda ülesandes 204 kirjutatud muutujale `paarisNegArv` samaväärne definitsioon loenduriga.

2. Ehitajad. Teatud juhtudel on otstarbekas ehitada liste akumulaatoris, et vältida listi koostamist pikematest valmis juppidest, mis omakorda on koos-

tatud valmis juppidest jne, tekitades ümbertõstmiste laviini. Akumulaatoris satub iga element kohe oma õigele kohale.

Vaatame näiteks, kuidas võiks programmeerida listi ümberpööramise; vastav funktsioon on moodulis Prelude muutuja `reverse` väärtuseks. Defineerime siin ta muutujasse `tagurpidi`, mille signatuur on

```
tagurpidi
  :: [a] -> [a]
```

Otsese rekursiooniga tehes saame definitsiooni

```
tagurpidi (x : xs)
  = tagurpidi xs ++ [x]
tagurpidi _
  = []
```
(99)

Arvutades selle järgi, toimub mittetühja listi juhul rekursiivse pöördumise tulemuse kopeerimine listi pea ette. Kopeeritava listi pikkus on esimesel korral 1, teisel 2 jne võrra väiksem algse listi pikkusest. Kogu arvutuse maht on seetõttu võrdeline ümberpööratava listi pikkuse ruuduga.

Vaatame aga definitsiooni

```
tagurpidi xs
  = let
      tagur (x : xs) as
        = tagur xs (x : as)
      tagur _        as
       = as
    in
    tagur xs []
```
(100)

Lokaalse operaatori `tagur` argument `as` on akumulaator, kus ehitatakse tulemuslist. Definitsiooni (100) järgi arvutades toimub igal rekursioonisammul vaid ühe elemendi ümberpaigutamine ühe listi algusest teise algusse. Listi koostamine talle elemente algusse lisades tähendab teisi sõnu tema koostamist lõpust alguse poole. Seega lõpptulemusena ongi seal elemendid algsega vastupidises järjekorras.

Arvutuse töömaht on (100) korral võrdeline ümbertõstetavate elementide arvuga ehk algse listi pikkusega. Akumulaatori kasutamine andis olulise võidu töökiiruses.

Kui otsese rekursiooni korral oleksid rekursiivsete pöördumiste antavad vahetulemused paarid, siis tasub samuti proovida akumulaatoritehnikat. Siis

saab paari asemel kasutada kaht eraldi lisaargumenti ning paari moodustada alles lõpus, baasjuhul. Sellega säästetakse ressurss, mis muidu kulub vaid ajutiselt vajalike paaride moodustamisele.

Varem koodiga (73) või (74) defineeritud operaatori `kaheksLahuta` saab sarnaselt defineerida ka akumulaatoritehnikaga: tulemuseks oleva paari komponentlistid saab koostada akumulaatoris.

Võttes kummagi listi jaoks kasutusele lisaargumendi, saame koodi

```
kaheksLahuta x xs
  = let
      lahuta as bs (z : zs)
        = if z < x
            then lahuta (z : as) bs zs
            else lahuta as (z : bs) zs
      lahuta as bs _
        = (reverse as , reverse bs)
    in
    lahuta [] [] xs
```
(101)

Selle järgi arvutades on algselt akumulaatorid tühjad ning töö käigus pannakse ühte neist lahkmest väiksemad ja teise ülejäänud andmed. Kuna sellise arvutusviisi korral salvestuvad andmed listidesse originaaliga võrreldes vastupidises järjestuses, nagu juba nägime, tuleb varasemate definitsioonidega samaväärsuse säilitamiseks akumulaatorid enne väljaandmist ümber pöörata.

Vajadus akumulaatorid enne väljaandmist ümber pöörata võib olla paaridest loobumise hind; sellele kulubki enam-vähem niisama palju aega kui paaride moodustamisele (väike võit või kaotus sõltub Haskelli realisatsioonist). Muidugi on see ajakulu tühine võrreldes variandiga, kus elemendid jooksvalt lisataksegi listi lõppu, nagu nägime näites (99).

Kuid selge tagasilöögi annab kood (101) lõpmatu argumentlisti puhul: arvutus jääbki tööle ilma midagi välja andmata. Paari ei teki, sest see moodustataks alles rekursiooni baasjuhul, kuhu arvutus ei jõua. Varasemate definitsioonide (73) ja (74) järgi arvutamine annab samasugusel argumendil korraliku vastuse; ainsaks puuduseks võib lugeda asjaolu, et kui kas lahkmest väiksemaid või ülejäänuid on vaid lõplik arv, siis tulemuspaari vastavasse komponenti tekib osaline, mitte lõplik list. Definitsioon (101) pole definitsioonidega (73) ja (74) täiesti samaväärne.

Ülesandeid

241. Defineerime muutuja `sabad` koodiga

```
sabad xs
  = reverse (tails xs)'
```

st tema väärtuseks on funktsioon, mis võtab argumendiks listi l ja kui see on lõplik, siis annab tulemuseks listi, milles on l kõik lõpuosad pikkuse kasvamise järjekorras.

Selle definitsiooni järgi arvutades moodustub vahestruktuur — algse listi alamlistide list pikkuse kahanemise järjekorras. Kirjutada muutujale `sabad` samaväärne definitsioon, mille järgi arvutades vahestruktuure ei teki.

242. Defineerida muutuja `kahend` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see on mittenegatiivne, siis annab tulemuseks listina arvu n kahendesituse numbrid ilma algusnullideta tavalises järjekorras. Negatiivsel argumendil väärtustamine peab lõppema adekvaatse veateatega. Vältida lististruktuuri korduvat ehitamist.

3. Arvutajad. Üldiselt kogub akumulaator mistahes arvutuste vahetulemusi. Ka loendurid ja ehitaja-akumulaatorid põhimõtteliselt teevad sedasama.

Parameeter `a` geomeetrilisi jadu defineerivas koodis (87) ja parameeter `n` Collatzi jadu defineerivas koodis (94) on ehtsad arvutajad-akumulaatorid. Nad olid küll mitte programmeerija omalooming, vaid juba ülesande püstituses sees.

Uue näitena kirjeldame akumulaatoritehnikaga lõpmatu listi kõigist Fibonacci arvudest. Et iga liikme arvutamiseks on vaja kaht eelmist, toome sisse kaks akumulaatorit, mis tähistavad jada jooksvat kaht järjestikust liiget. Koodiks sobib

```
fibs
  = let
      fibs a b
        = a : fibs b (a + b)
    in
    fibs 0 1
```
(102)

Siin akumulaator `a` hoiab seda Fibonacci arvu, mis jooksval rekursioonisammul listi kirjutatakse, akumulaator `b` järgmist. Igal rekursiivsel pöördumisel nihkub teine argument esimese kohale ja uueks teiseks argumendiks tuleb argumentide summa ehk järjekorras järgmine Fibonacci arv.

Nagu samaväärne listirekursiooniga definitsioon (92), annab ka akumulaatoritega definitsioon (102) lineaarse tööajaga arvutuse.

Arvutaja- ja ehitaja-akumulaatori kontseptsioonid on praktiliselt identsed, sest kui arvutuse käik akumulaatori jooksvat väärtustamist ei nõua, siis akumulaator kogub endasse vaid tehteid, ehitab pikka väärtustamata avaldist.

Ülesandeid

243. Lahendada ülesanded 220 ja 223 akumulaatoritega.

4. Sabarekursioon. Akumulaatoritehnika viib tihti nn sabarekursiivse definitsioonini. Sabarekursiivseks nimetatakse funktsiooni rekursiivset definitsiooni, mille järgi arvutades antakse rekursiivse pöördumise tulemus alati ilma mingite vaheteisendusteta ka kogu arvutuse tulemuseks.

Lokaalse muutuja `arv` definitsioonis koodis (98) on ainus rekursiivne pöördumine esimese deklaratsiooni paremas pooles ja pöördumise tulemus on ühtlasi kogu selle juhu tulemuseks. Võrdluseks näiteks muutuja `nullideArv` definitsiooni (72) korral kui listi pea on 0, siis liidetakse rekursiivse pöördumise tulemusele veel 1. Seega `arv` definitsioon koodis (98) on sabarekursiivne, aga definitsioon (72) ei ole.

Samuti on koodis (100) lokaalne akumulaatoriga definitsioon sabarekursiivne, otsese rekursiooniga definitsioon (99) aga mitte.

Sabarekursiooni tähtsus seisneb selles, et sabarekursiivseid definitsioone on võimalik kompileerimisel optimeerida. Kui rekursiivse pöördumisega on jooksva rekursioonitaseme töö sisuliselt lõppenud, võib rekursiivsel pöördumisel jooksva taseme lokaalsete muutujate väärtused järgmise rekursioonitaseme väärtustega üle kirjutada. Nii tehes tekib funktsionaalsest rekursiivsest definitsioonist imperatiivne tsükliga algoritm.

Ülesandeid

244. Selgitada, miks definitsioonid (63), (67), (77) pole sabarekursiivsed.

4.2.2 Järjehoidjad

Järjehoidjaks nimetame abiparameetrit, mille ülesandeks on mingis mõttes tööjärje meelespidamine, kuid mis lõpptulemuse väljaandmisel enam rolli ei mängi, töö lõpul ta unustatakse ära. Tavaliselt toimub järjehoidjates ka mingi arvutamine, need on ühtlasi akumulaatorid.

Vastanduvaks võrdluseks, akumulaatorid näidetes (98) ja (100) kandsid lõpptulemust, definitsioonide baasjuhus anti akumulaator ise välja.

1. Lihtsad järjehoidjad. Lihtsamal juhul võib järge hoida suvaliste kättesattuvate andmetega ja akumuleerimist ei toimu.

Kirjeldame protseduuri, mis kuulab standardsisendist sümboleid, kuni sisestatakse mõnd sümbolit kaks korda järjest, siis lõpetab; signatuur olgu

```
kuulaKorduseni
  :: IO ()
```

.

Siin on põhiprobleemiks, kuidas osata otsustada, kas sümbol on eelmisega võrdne või mitte. Iga sümbolit tuleb meeles hoida, kuni järgmine sümbol on sisse loetud.

Toome sisse lokaalse operaatori `kuula`, mis saab argumendiks viimati loetud sümboli. Selle operaatori poole pöördumiseks peab üks sümbol olema juba kuulatud, see tuleb teha põhiprotseduuris. Tulemuseks võib olla definitsioon

```
kuulaKorduseni
  = let
      kuula d
        = do
            c <- getChar
            if c == d
              then putStrLn ""
              else kuula c
    in
    do
      hSetBuffering stdin NoBuffering
      c <- getChar
      kuula c
```

. (103)

Ülesandeid

245. Anda muutujale `kuula01` väärtuseks protseduur, mis loeb standardsisendist sümboleid, kuni sealt tulevad järjest sümbolid 0 ja 1, ja lõpetab siis töö.

2. Loenduri tüüpi järjehoidjad.

2.1. Tsükliindeksid. Järjehoidja võib mängida imperatiivse keele tsükliindeksi rolli, näidates, kui kaugel arvutus omadega on.

Neid läheb tihti vaja listide defineerimisel, kui elemendid ei avaldu vaid eelmise või eelnevate kaudu, vaid sõltuvad ka järjekorranumbrist.

Geomeetriliste jadade definitsioonis (87) ja Collatzi jadade definitsioonis (94) polnud tsükliindeksit vaja, sest nende jadade iga järgmine liige avaldub eelneva kaudu ühtemoodi sõltumata positsioonist jada alguse suhtes.

Vaatame olukorda, kus see nii ei ole. Koodiga (63) defineerisime funktsiooni, mis mittenegatiivsel täisarvulisel argumendil n annab tulemuseks summa $1^4 + \ldots + n^4$;

püüame nüüd defineerida listi järjekorras kõigi selliste summadega. Signatuur olgu

```
suuredSummad
  :: (Integral a).
  => [a]
```

Siin tuleb igal sammul liita just selle sammu järjekorranumbri 4. aste. Võttes jooksva summa salvestamise jaoks kasutusele akumulaatori `a` ja sammunumbri jaoks loenduri `i` — tsükliindeksi —, saame koodi

```
suuredSummad
  = let
      summad a i
        = a : summad (a + i ^ 4) (i + 1) .
    in
    summad 0 1
```
(104)

Algselt (abioperaatori esimesel väljakutsel) on akumulaatori väärtus 0 ja tsükliindeksi väärtus 1. Igal rekursiivsel pöördumisel kirjutatakse akumulaatori jooksev väärtus listi, akumulaatorisse lisatakse jooksva tsükliindeksi 4. astme liitmine ja tsükliindeksile liidetakse 1. Niimoodi tekivad akumulaatorisse 1^4 liitmine, 2^4 liitmine jne ning tulemuslist moodustub nende tehete tulemustest.

Isegi kui vaja on vaid üksikuid elemente ja list tervikuna ei paku huvi, võib olla mõttekas algul defineerida list. Kui testimine on näidanud, et see definitsioon on õige, siis on kerge vaevaga võimalik definitsioon ümber töötada nii, et ta arvutaks mingit konkreetset listi elementi ilma listi ennast koostamata. Selleks tuleb akumulaatori salvestamine listi ära jätta, lisada aga lõpetamistingimus. Arvutuspõhimõte säilib.

Tõsine alternatiiv on muidugi pärast lõpmatu listi defineerimist hoopis lisada definitsioon, mis sealt listist elemente loeb. See on eriti mõistlik siis, kui samasugust arvutust oleks vaja korduvalt käivitada: listist elementide lugemine on arvatavasti kiirem kui iga kord otsast peale arvutamine. See võib aidata ka koodi loetavust parandada.

Näiteks võime deklaratsiooni (104) modifitseerides anda koodiga (63) defineeritud muutujale `suurSumma` uue samaväärse definitsiooni, mis vahelisti ei kasuta. Võrreldes definitsiooniga (104) peame lisama välise parameetri `n`, mis näitab otsitava summa järjekorranumbrit, ja tema negatiivse väärtuse käsitlemise eraldi; salvestamine listi tuleb ära jätta, kuid lisada tuleb hargnemine lokaalses deklaratsioonis, et juhul, kui tsükliindeksi väärtus ületab `n` oma, arvutus lõpetataks ja antaks akumu-

laator välja. Tekib kood

```
suurSumma n
  | n >= 0
    = let
        summa a i
          | i <= n
            = summa (a + i ^ 4) (i + 1)
          | otherwise
            = a
      in
      summa 0 1
  | otherwise
    = error "suurSumma: neg. liidetavate arv"
```
. (105)

Teise näitena samast vallast oletame, et meid ei rahulda üksikute Fibonacci arvude arvutamine listist lugemise põhimõttel koodiga (93). Töötades ümber Fibonacci arvude listi akumulaatoritega definitsiooni (102), saame koodi

```
fib n
  | n >= 0
    = let
        fib a _ 0
          = a
        fib a b i
          = fib b (a + b) (i - 1)
      in
      fib 0 1 n
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
. (106)

Võrreldes koodiga (102) on tehtud samad muudatused mis koodis (105) sai tehtud võrreldes koodiga (104). Võib märgata, et loenduril `i` pole peale tsükliindeksi rolli siin muud tähendust: ta pole nt parajasti akumulaatoris oleva Fibonacci arvu järjekorranumber, sest akumulaatorites arvutatakse neid järjekorranumbri kasvades, kuid tema väärtus muutub kahanevalt (selline suunavalik on tehtud programmeerimise mugavuse pärast).

Arvutus definitsiooni (106) järgi teeb tööd võrdeliselt argumendi väärtusega. See peaks toimuma isegi pisut kiiremini kui definitsiooni (68) korral, sest definitsiooni (68) puhul konstrueeritakse igal rekursioonisammul paar, mis nõuab lisaressurssi nii nagu listi moodustaminegi. Eelis arvutuskiiruses on siiski vaid marginaalne.

Nii koodis (105) kui ka (106) on lokaalsed definitsioonid sabarekursiivsed.

Valikul, kas lõpmatu listi definitsiooni kasutada vaid vaheetapina funktsiooni definitsiooni kirjutamisel või realiseeridagi funktsiooni definitsioon listist lugemisena, tuleb arvestada ka seda, et listist lugeminegi võtab aega võrdeliselt elemendi järjekorranumbriga listis.

Kindlasti pole mõtet programmeerida ruutfunktsiooni listist lugemise kaudu isegi mitte täisarvude korral, kuigi täisarvude ruutude list on kergesti kirjeldatav komprehensioonsüntaksiga (46), sest konkreetse arvu ruudu leiab vaid ühe tehtega.

Kummatigi, kui siiski vajatakse järjest kõiki täisarvude ruute, võib mõelda komprehensioonsüntaksi asemel akumulaatoritele ja kirjutada kood

```
sqrs
  :: (Integral a),
  => [a]

sqrs
  = let
      sqrs a d
        = a : sqrs (a + d) (d + 2)
    in
    sqrs 0 1
```
(107)

See kasutab tuntud fakti, et täisarvude ruudud on saadavad järjestikuste paaritute arvude liitmisel: akumulaator `a` hoiab jooksvat ruutarvu ja tsükliindeksi moodi `d` jooksvat paaritut arvu.

Kood (107) on pisut efektiivsem võrreldes komprehensiooniga lahendusega (46), sest viimasel juhul tekib vaheandmestruktuur — kõigi naturaalarvude list —, pealegi kasutab kood (107) ainult liitmisi, samas kui (46) nõuab korrutamist.

Ülesandeid

246. Defineerida muutuja `facs` väärtuseks list, mille elementideks on järjest kõigi naturaalarvude faktoriaalid. Listi algusosa väljaarvutamine kulutagu aega võrdeliselt arvutatud osa pikkusega.

247. Defineerida muutuja `kolmnurgad` väärtuseks list, mille elementideks on järjest kõik `kolmnurkarvud` ehk binoomkordajad C_{n+1}^2, $n \in \mathbb{N}$.

248. Defineerida muutuja `madal` väärtuseks predikaat, mis võtab argumendiks arvude listi ja annab väärtuseks True, kui ükski element seal pole negatiivne ega suurem oma positsiooninumbrist (loeme alates 0-st), muidu False.

249. Anda koodiga (65) defineeritud muutujale `kahFac` samaväärne definitsioon, mille järgi toimuks arvutamine akumulaatoris. Saada läbi kahe parameetriga.

250. Kirjutada koodiga (71) defineeritud muutujale `sq2` uus samaväärne definitsioon, mille korral arvutus toimub akumulaatorites ja vahestruktuure ei moodustata.

251. Kirjutada ülesande 186 uus lahendus, mille korral arvutus toimub akumulaatorites ja vahestruktuure ei moodustata.

2.2. Teistsugused loendurid. Loendur ei pruugi muutuda vaid ühes suunas. Kui on vaja valikut teha kindla lõpliku arvu võimaluste vahel, piisab meeles pidada vaid loenduri jääki teatud arvuga jagamisel.

Defineerime muutuja `tasakaal` väärtuseks funktsiooni, mis võtab argumendiks sõne ja annab tulemuseks tõeväärtuse vastavalt sellele, kas sulud selles on tasakaalus või mitte; signatuur on

```
tasakaal
  :: String -> Bool
```

Toome sisse lokaalse samanimelise abioperaatori lisaargumendiga `n`, mille otstarve on sõne lugemisel meeles hoida jooksvat "sulusügavust" ehk loendada alustatud, kuid lõpetamata sulupaare. Sulud on tasakaalus parajasti siis, kui sügavus on sõne lõpul 0 ja pole kuskil negatiivne (lõpetavaid sulge rohkem kui avavaid). Sobib kood

```
tasakaal s
  = let
      tasakaal n (c : cs)
        = let
            d = case c of
                  '('
                    -> 1
                  ')'
                    -> -1
                  _
                    -> 0
          in
          n >= 0 && tasakaal (n + d) cs
      tasakaal n _
        = n == 0
    in
    tasakaal 0 s
```
(108)

Koodi (108) lokaalses definitsioonis esineb rekursiivne pöördumine operaatori && teise argumendina. Sellegipoolest võib seda definitsiooni sabarekursiivseks lugeda, sest operaatori && teine argument muutub oluliseks alles siis, kui esimene on tõeseks väärtustatud, ja sel juhul tagastatakse teine argument ilma teisendusteta.

Pöördume veel tagasi muutuja `kaheksLoe` ülesande juurde, millele oleme näinud juba kolme lahendust: topeltsammudega (76), argumentide vahetamisega (77) ning vastastikrekursiooniga (80). Et probleemiks on meeles pidada, kumba listi tuleb panna järjekordne element, on üks loomulik lähenemine kasutada järjehoidjat. Kahe võimaluse eristamiseks sobib tõeväärtusetüüpi järjehoidja, sellega saame koodi

```
kaheksLoe xs
  = let
      loe b (x : xs)
        = let
            (us , vs)
              = loe (not b) xs
          in
          if b
            then (us , x : vs)
            else (x : us , vs)
      loe _ _
        = ([] , [])
    in
    loe False xs
```
. (109)

Lokaalse operaatori `loe` argumendi `b` väärtuse False korral lisatakse jooksev element vasakusse listi, väärtuse True korral paremasse listi. Igal rekursiivsel pöördumisel muudetakse tõeväärtusparameetri väärtus vastupidiseks. Järjehoidja `b` niisiis akumuleerib eitusi, mis on loendurilaadne käitumine (lihtsalt igal teisel sammul läheb uuesti nulli).

Ülesandeid

252. Anda ülesandes 204 defineeritud muutujale `paarisNegArv` koodi (109) eeskujul samaväärne definitsioon tõeväärtusetüüpi järjehoidjaga.

253. Kirjutada muutujale `üleÜheEtte` definitsiooni (96) eeskujul alternatiivne definitsioon, kus lokaalne operaator `kaheksLoe` oleks defineeritud koodi (109) eeskujul järjehoidjaga.

254. Defineerida muutuja `dropÜksus` väärtuseks funktsioon, mis võtab argumendiks kaks sümbolit ja sõne ja annab tulemuseks sõne lõpuosa, mis järgneb esimesele sellisele kohale, kus teist sümbolit on esinenud rohkem kui esimest; kui niisugust kohta pole, siis peab arvutus lõppema veateatega.

255. Kas abimuutuja `loe` definitsioon koodis (109) on sabarekursiivne?

3. Ehitaja tüüpi järjehoidjad. Kui on vaja meeles hoida teadmata hulgal andmeid, tuleb nad koondada andmestruktuuri. Selline argument on põhimõtteliselt akumuleeriv järjehoidja.

Üks olukord, kus see on vältimatu, on potentsiaalselt lõpmatu listi elemendipaaride (või ka suuremate komplektide) ammendava läbivaatuse tegemine. Lisaargumendis tuleb hoida neid elemente, millest moodustatud paarid on kõik juba läbi vaadatud.

Olgu meil näiteks vaja mistahes listi kohta kontrollida, ega temas ei ole korduvaid elemente; ootame definitsiooni vastavalt signatuurile

```
kordusteta
  :: (Eq a)     .
  => [a] -> Bool
```

Otsese rekursiooniga saaks programmeerija kirjutada definitsiooni

```
kordusteta (x : xs)
  = not (elem x xs) && kordusteta xs
kordusteta _                          .
  = True
```
(110)

Kui argumendiks on tühi list, siis tuleb selle koodi järgi väärtuseks True, mis on õige, sest tühjas listis ükski element ei kordu. Mittetühja listi puhul peab elementide mittekorduvuseks olema täidetud parajasti kaks tingimust: esiteks, et esimene element ei esine järgmiste seas, ja teiseks, et järgmiste elementide seas pole korduvaid. Nende tingimuste kontrolli eest hoolitsevad vastavalt `not (elem x xs)` ja rekursiivne pöördumine `kordusteta xs`. Seega võib tunduda, et kood (110) vastab tingimustele.

Paraku lõpmatul listil, milles esineb kordusi, kuid esimene element on unikaalne, arvutus definitsiooni (110) järgi kordusi üles ei leia, sest püüab kõigepealt esimest elementi kõigi järgnevatega võrrelda ja jääb lõpmatult tööle.

Algoritm, mis siinkohal aitaks, peaks tegema võrdlusi teises järjekorras. Iga elementi tuleks võrrelda temast eespool olevatega, mitte tagapool olevatega nagu teeb (110). Igale elemendile eelneb listis vaid lõplik arv elemente ja seetõttu jõuaks niisugune algoritm suvalise kahe elemendi võrdlemiseni lõpliku arvu võrdluste järel. Niisuguse tööskeemi korral oleks listi iga elemendini jõudmisel kõik eelnevad juba omavahel läbi võrreldud.

Võtame lokaalses definitsioonis kasutusele listitüüpi lisaargumendi, mis arvutuse käigus sisaldab just neid elemente originaallisti algusest, mis hetkeseisuga on omavahel läbi võrreldud. Algul pole ühtki elementi analüüsitud, mistõttu algsel väljakutsel anname selleks argumendiks tühja listi. Töö käigus tuleb kontrollida iga

järgneva elemendi esinemist abilistis: kui esineb, siis on tuvastatud kordus algses listis, vastasel korral tuleb lisada temagi järgneval rekursiivsel pöördumisel abilisti. Selle algoritmi realiseerib kood

```
kordusteta xs
  = let
      abi as (z : zs)
        = not (elem z as) && abi (z : as) zs
      abi _  _
        = True
    in
    abi [] xs
```
. (111)

Arvutus selle definitsiooni järgi leiab korduse üles isegi juhul, kui list on osaline.

Elemendid salvestuvad abilisti vastupidises järjekorras võrreldes originaallistiga, kuid antud ülesandes see vahet ei tee.

Ülesandeid

256. Defineerida muutuja `kuulaKorduseni'` väärtuseks protseduur, mis kuulab standardsisendist sümboleid, kuni mingi sümbol kordub (mitte tingimata järjest) ja lõpetab siis töö.

257. Defineerida muutuja `kuula01JaEdasta` väärtuseks protseduur, mis loeb standardsisendist sümboleid, kuni sealt tulevad järjest sümbolid 0 ja 1, ja lõpetab siis töö, edastades sõne, milles on õiges järjekorras kõik loetud sümbolid.

258. Defineerida muutuja `kümnendesitus` väärtuseks funktsioon, mis võtab argumendiks ratsionaalarvu q ja kui see on mittenegatiivne, siis annab tulemuseks q kümnendesituse sõnekujul, kus võimalik lõpmatult korduv periood on esitatud sulgudes (näiteks $q = \frac{1}{22}$ korral peaks vastuseks olema "0.0(45)"). Negatiivsetel argumentidel lõpetagu arvutus sobiva veateatega.

259. Cantori hulk on reaalarvude hulk, kuhu kuuluvad parajasti need reaalarvud, mille murdosa kolmendesituses ei esine numbrit 1 (esitused, mis lõpevad 2-ga perioodis, ei tule arvesse).

    Defineerida muutuja `cantoriKübe` väärtuseks funktsioon, mis võtab argumendiks ratsionaalarvu q ja annab väärtuseks True või False vastavalt sellele, kas q kuulub Cantori hulka või mitte.

260. Defineerida muutuja `disjunktsed` väärtuseks karritatud funktsioon, mis võtab argumendiks kaks listi ja kui kumbki neist on lõplik või lõpmatu ja

nad sisaldavad ühiseid elemente, siis annab välja tõeväärtuse False, kui aga mõlemad listid on lõplikud ja ühiseid elemente ei sisalda, annab välja True. Mis juhtub, kui üks listidest on osaline?

261. Defineerida muutuja `uuriKolmikud` väärtuseks funktsioon, mis võtab argumendiks täisarvulisti ja annab tulemuseks True, kui listis leidub kolm erinevas positsioonis elementi, mis moodustavad aritmeetilise progressiooni, vastasel korral kui list on lõplik, siis annab välja False.

262. Kui definitsioonis (111) konjunktsiooni pooled ära vahetada, kas definitsioon muutuks halvemaks, paremaks või pole vahet?

4. Arvutaja tüüpi järjehoidjad. Need on muidugi kõige levinumad.

Olgu meil vaja defineerida muutuja `nullsummadeArv` väärtuseks funktsioon, mis võtab argumendiks listi ja annab välja arvu, kui palju on sel listil mittetühje algusjuppe, mille elementide summa on null. Signatuur peaks olema

```
nullsummadeArv
  :: (Num a)   .
  => [a] -> Int
```

Püstitus on sarnane muutuja `nullideArv` omaga, kus loendati listis elemendina esinevaid nulle. Nüüd on vaja listi algusest järjest arve kokku liita ja lugeda nulliga võrduvaid vahesummasid.

Teeme ümber muutuja `nullideArv` loendurit kasutava definitsiooni (98), lisades lokaalsele operaatorile veel ühe lisaparameetri `s`, mis hoiab jooksvat summat. Uue summa, kus ka listi järgmine element on sisse arvestatud, seome lokaalse muutujaga `s'`. Saame koodi

```
nullsummadeArv xs
  = let
      arv s n (z : zs)
        = let
            s'
              = s + z
          in                                                ,
          arv s' (if s' == 0 then n + 1 else n) zs
      arv _ n _
        = n
    in
    arv 0 0 xs
```

(112)

kus lokaalse operaatori `arv` definitsioon on jällegi sabarekursiivne.

Sarnase näitena olgu meil vaja defineerida muutuja `maxArv` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks tema maksimaalsete elementide arvu, signatuuriga

```
maxArv
  :: (Ord a)    .
  => [a] -> Int
```

Muidugi on võimalik kirjutada kood, millega kõigepealt vaadatakse järele, milline element on maksimaalne, ja seejärel loetakse nad kokku, kuid see tähendaks listi kahekordset läbivaatamist. Programmeerime ühekordse läbivaatusega algoritmi, mis kasutab jooksvat maksimumi (listi algusest lugedes) hoidvat järjehoidjat ja nende loendurit. Kui leitakse senisest maksimumist suurem element, läheb see järgmisel sammul järjehoidjaks ja loendur pannakse 1-ks; kui leitakse senise maksimumiga võrdne element, siis kasvatatakse loendurit; muidu jäävad abiparameetrid muutmata. Siis on lõpuks, kui list on läbi, loenduris kogu listi maksimaalsete elementide arv, mida oligi vaja arvutada.

Kuna tühjal listil maksimaalsest elemendist rääkida ei saa, siis tuleb see juht eraldi läbi vaadata. Mittetühja listi puhul on esimene jooksev maksimum (järjehoidja algväärtus) listi esimene element, nende jooksev arv (loenduri algväärtus) on 1. Kirjeldatud ideed realiseerib kood

```
maxArv (x : xs)
  = let
      arv m i (z : zs)
        = case compare m z of
            GT
              -> arv m i zs
            EQ
              -> arv m (i + 1) zs
            _
              -> arv z 1 zs
      arv _ i _
        = i
    in
    arv x 1 xs
maxArv _
  = 0
```
. (113)

Lokaalse operaatori `arv` definitsioon koodis (113) on järjekordne näide sabarekursioonist: kõik rekursiivsed pöördumised, olgugi et eri alternatiivides, on viimased operatsioonid, mida antud rekursioonitasemel tehakse.

Ülesandeid

263. Defineerida muutuja `sumProdVõrdseteArv` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab tulemuseks tema selliste algusjuppide arvu, mille elementide summa ja korrutis on võrdsed.

264. Defineerida muutuja `maxKonst` väärtuseks funktsioon, mis võtab argumendiks võrdusega tüüpi elementide listi ja annab väärtuseks selle listi pikima konstantse (st võrdsete elementidega) lõigu pikkuse.

265. Defineerida muutuja `maxJärjest` väärtuseks funktsioon, mis võtab argumendiks võrdusega tüüpi elementide listi l ja annab tulemuseks listi, mis koosneb l neist elementidest, mida listis kõige rohkem järjest esineb, leitud maksimumpikkusega lõikude esinemise järjekorras.

4.2.3 Jooksev väärtustamine

Eelnevas sai juba märgitud, et akumulaator kogub vahetulemusi üldiselt väärtustamata kujul. See tähendab, et mälutarbe seisukohalt on akumulaatoritega definitsioonid niisama kehvad kui sellised otsese rekursiooniga definitsioonid, kus vahetulemuste leidmine enne baasjuhu töötlemist üldse võimalik ei ole, kuna üheks operandiks on rekursiivse pöördumise tulemus.

Operaatori `suurSumma` algse definitsiooni (63) järgi arvutamisel ei saa liitmistehet sooritada enne, kui rekursiivne pöördumine on andnud tulemuse. Seetõttu on paratamatu, et rekursiivsete pöördumiste jada kasvatab mällu vaid pikka väärtustamata avaldist. Sisuline arvutamine saab toimuda alles tagasipöördumisel.

Uues definitsioonis (105) asub pluss akumulaatoris, mis põhimõtteliselt võimaldab liitmise ära teha enne rekursiivset väljakutset. Akumulaatori jooksev väärtustamine väldiks vahetulemuste hoidmiseks vajamineva mälu piiramatut kasvu. Rekursiooni baasini jõudmisel oleks lõppvastus juba käes.

Reaalselt aga on mäluhõive definitsiooniga (105) arvutades niisama suur kui definitsiooni (63) puhul, sest väärtustamine on vaikimisi laisk ja ka akumulaatoris ei tehta liitmisi enne kui alles lõpus.

Näeme, et laisal väärtustamisel on peale soovitud omaduse, et mittevajalike asjade arvutamisele ei kulu aega, ka teine ja üsna ebameeldiv tagajärg.

1. Agaraks sundimine. Haskellis on võimalik väärtustamist kohati agaraks pöörata, võttes appi agara rakendamise operaatori `$!`.

Infiksoperaator `$!` on paremassotsiatiivne prioriteediga 0, nii nagu operaator `$`. Tema väärtuseks on funktsioon, mis võtab argumendiks funktsiooni f ja tema potentsiaalse argumendi x: kui $x \neq \bot$, siis annab väärtuseks $f\,x$, kui aga $x = \bot$, siis annab väärtuseks $\bot$.

Idee on, et kujul $\mathfrak{f}$ `$!` $\mathfrak{e}$ oleva avaldise arvutamisel väärtustatakse kõigepealt avaldist $\mathfrak{e}$ senikaua, kuni saadakse selline vahetulemus $\mathfrak{e}'$, mille kuju näitab, et väärtus on normaalne, misjärel väärtustatakse $\mathfrak{f}\ \mathfrak{e}'$.

Eespool on mainitud, et `const` $\mathfrak{a}$ on laisk funktsioon iga argumendi $\mathfrak{a}$ korral, mistõttu näiteks avaldiste `const 0 undefined` ja `const 0 $ undefined` väärtus on 0, nende väärtustamine lõpeb normaalselt.

Seevastu avaldise `const 0 $! undefined` väärtus on $\bot$, tema väärtustamine lõpeb veateatega, mille annab `undefined`.

Sarnaselt jääb avaldise `const 5 $! length [1 .. ]` väärtustamine lõpmatusse tsüklisse, ehkki avaldise `const 5 $ length [1 .. ]` väärtustamine lõpeb normaalselt vastusega `5`.

Pangem tähele, et `$!` teist argumenti ei väärtustata tingimata lõpuni. Üldiselt piisab väärtuse normaalsuse selgumiseks tema moodustava konstruktori kättesaamisest. (Teisiti on funktsioonide ja protseduuride korral, sest neid ei moodustata konstruktoritega.)

Näiteks avaldise `const 5 $! (undefined , undefined)` väärtus on 5, sest paarikonstruktori `,` ilmumine näitab, et väärtus on normaalne, ja viga ei teki.

Muuhulgas kui argumendi väärtus on mittetühi list, siis teda väärtustatakse vaid esimese kooloni ilmumiseni, pea ja saba jäävad väärtustamata.

Näiteks `head $! [1 .. ]` ja `head [1 .. ]` annavad sama tulemuse 1, sest mõlemal juhul väärtustatakse argument kuni esimese kooloni ilmumiseni ja mitte kaugemale, nii et lõpmatut arvutust ei teki.

2. Väärtustusjärjekorra ohje. Akumulaatorite jooksvaks väärtustamiseks tuleb operaator `$!` lisada rekursiivsetesse pöördumistesse nii, et akumulaatorile rakenduv avaldis saaks tema esimeseks (vasakpoolseks) ja akumulaator ise teiseks (parempoolseks) argumendiks.

Koodi (105) lokaalse definitsiooni akumulaatori jooksvaks väärtustamiseks peame esimese valvuri juhus sundima `summa` rakenduse argumendile `a + i ^ 4` agaraks. Seega operaatori `$!` esimene argument peab olema `summa` ja teine

`a + i ^ 4`. Siin osutub mugavaks prefiksne rakendamine, sellega saame koodi

```
suurSumma n
  | n >= 0
    = let
        summa a i
          | i <= n
            = ($!) summa (a + i ^ 4) (i + 1)
          | otherwise
            = a
      in
      summa 0 1
  | otherwise
    = error "suurSumma: neg. liidetavate arv"
```
. (114)

Tänu agarale väärtustamisele saab definitsiooni (114) korral muutujat `suurSumma` edukalt kasutada mitu suurusjärku suuremate argumentidega kui varasemate definitsioonide puhul.

Järgmist argumenti `i + 1` see `$!` väärtustama ei sunni, kuid seda polegi vaja. See avaldis väärtustatakse rekursiooni järgmisel tasemel niikuinii, sest tema väärtuse järgi on vaja valida haru.

Ka muutuja `fib` definitsiooni (106) võime samasuguse lisandusega paremaks teha, kirjutades

```
fib n
  | n >= 0
    = let
        fib a _ 0
          = a
        fib a b i
          = ($!) fib b (a + b) (i - 1)
      in
      fib 0 1 n
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
. (115)

Definitsioonis (115) ei mõju operaator `$!` küll liitmist sisaldavale avaldisele otse, kuid siin piisabki ainult esimese akumulaatori jooksvast väärtustamisest. Kui `a` on väärtustatud ja `b` sunnitakse väärtustama rekursiivsel pöördumisel, siis argument `a + b` sisaldab ainult ühe liitmistehte ning järgmisel rekursioonitasemel on `a` jällegi väärtustatud. Seega avaldise piiramatut pikenemist ei toimu.

Kui soovime listi jooksvalt lõpuni välja arvutada, tuleb agar väärtustamine viia rekursiivselt lististruktuuri sisse. See käik garanteerib listi lõplikkuse.

Toome sisse operaatori `lõplik` signatuuri ja definitsiooniga

```
lõplik
  :: [a] -> [a]'

lõplik (x : xs)
  = ($!) (x :) (lõplik xs)
lõplik _                    .
  = []
```
(116)

Kujul `lõplik f` oleva avaldise väärtustamine selle definitsiooni põhjal põhimõtteliselt ehitab lõpuni avaldise `f` väärtuseks oleva listi struktuuri. See toimub tagant ettepoole: listi pead ei panda paika enne, kui saba on valmis, sabas omakorda esimest elementi ei panda kohale enne, kui järgmiste elementide list on valmis jne. Tulemusse ei ilmu ühtki konstruktorit enne, kui rekursioon on jõudnud põhjani ja ühtlasi argumentlisti struktuur on lõpuni välja arvutatud.

Seega niipea, kui on vaja mingilgi määral väärtustada avaldist kujul `lõplik f`, arvutatakse argumentlisti struktuur kohe lõpuni välja. Kui list on lõpmatu, jääb see protsess midagi välja andmata lõpmatult tööle.

Ülesandeid

266. Kirjutada oma moodulisse deklaratsioonid (63), (105), (114) ja neid kahekaupa välja kommenteerides kontrollida igaühe puhul, kui suure argumendiga on Hugs võimeline funktsiooni väärtust arvutama ilma mälu ületäitumiseta.

267. Kirjutada definitsiooni (115) teisend, mille korral ei jäetaks väärtustamata isegi mitte ühte liitmistehet.

268. Kirjutada ülesannete 249, 250, 251 lahendused nii, et arvutamise käigus pikki väärtustamata avaldisi ei tekiks.

269. Defineerida muutuja `sünna` väärtuseks karritatud funktsioon, mis võtab täisarvulised argumendid n ja k ning annab tulemuseks tõenäosuse, et kui teha k sõltumatut valikut samast n objektist, siis kõik k valitud objekti on paarikaupa erinevad. Näiteks `sünna 365 20` väärtuseks peab olema tõenäosus, et juhuslikult valitud 20 inimese hulgas ei leidu ühtki kaht, kelle sünnipäevad langeksid kokku (eeldusel, et ükski pole sündinud 29. veebruaril). Kui argumendid n ja k on sellised, mille korral antud ülesandepüstitus on mõttetu, peab arvutus andma olukorda täpselt kirjeldava veateate.

270. Mis on koodiga (116) defineeritud muutuja `lõplik` väärtus?

4.3 “Jaga ja valitse”

Strateegia “jaga ja valitse” tähendab, et mittetriviaalne ülesanne jagatakse igal rekursioonisammul kaheks või enamaks väiksemaks sama tüüpi alamülesandeks, mis lahendatakse rekursiivselt samal põhimõttel ja mille lahendustest kombineeritakse kokku terve ülesande lahendus. Triviaalsed juhud lahendatakse otse, ilma jagamata.

Mõnikord tuleneb selline lahendusviis loomuldasa ülesande püstitusest, mõnikord aga on see strateegia otstarbekas efektiivsuskaalutlustel. Kuid “jaga ja valitse” stiilis lahenduse otstarbekus sõltub paljudest asjaoludest ja alati selline arvutus kõige efektiivsem ei ole.

4.3.1 Vahetud “jaga ja valitse” realisatsioonid

1. Ühe mittetriviaalse alamülesandega olukorrad. Kui jaotuses sama tüüpi alamülesanneteks on alati ainult üks alamülesanne mittetriviaalne, siis sarnaneb programmeerimine senisega, sest igas harus on vaja ülimalt üht rekursiivset pöördumist. Lisandub vaid “jaga ja valitse” stiilis mõtlemine.

Olgu meil vaja muutuja `täisruut` väärtuseks saada funktsiooni, mis võtab argumendiks täisarvu tüübist Integer ja annab tulemuseks tõeväärtuse vastavalt sellele, kas see arv on täisruut; signatuur olgu

```
täisruut
  :: Integer -> Bool
```

Kuna ülipikkade arvude esitus ujukomaga on nii ebatäpne, et antud küsimus muutub mõttetuks, jäävad kõrvale variandid muutuja `sqrt` kasutamisega. Kuna aga ruutfunktsioon on kasvav, on võimalik klassikalisel nn lõigu poolitamise meetodil leida antud arvu ruutjuure täisosa ja kontollida, kas ta on täpne ruutjuur või mitte.

Lõigu poolitamise meetodi läbiviimiseks tuleb ette anda lõik, kuhu otsitav ruutjuur kindlasti kuulub. Tööpõhimõte on järgmine. Leiame täisarvu m võimalikult lõigu keskelt ja tõstame ta ruutu. Kui etteantud arv on saadud tulemusest väiksem, siis vastavalt on ka tema ruutjuur väiksem arvust m ehk kuulub alumisse osalõiku; seal otsime samamoodi edasi. Kui etteantud arv on arvu m ruudust suurem, siis on ka tema ruutjuur suurem arvust m ehk kuulub ülemisse osalõiku; siis otsime seal samamoodi edasi. Võib ka juhtuda, et etteantud arv ja katseväärtus võrduvad; siis oleme leidnud täpse ruutjuure. Kui oleme jõudnud lõiguni pikkusega 1, siis lahendame edasi läbiproovimisega.

Lõigu poolitamise meetod on üks kahendotsingu alaliik. Kahendotsinguks nimetatakse otsimismeetodit, kus otsinguala jaotatakse igal rekursioonisammul kaheks

võimalikult võrdseks osaks, määratakse otsitava kuuluvus ühte või teise ossa ja otsitakse selles samamoodi edasi. Kui igal rekursioonisammul õnnestub ala enam-vähem täpselt pooleks jagada, on rekursioonisammude arv ligikaudu võrdne ala suuruse logaritmiga. See annab tohutu aja kokkuhoiu võrreldes juhmi läbivaatusega.

Antud ülesandes tasub lõigu poolitamist kasutada ainult juhul, kui etteantud arv on 1-st suurem. Muidu on vastus niigi selge: argument on täisruut parajasti siis, kui ta on mittenegatiivne. Nii saame põhifunktsiooni definitsiooniks kirjutada

```
täisruut n
  | n > 1
    = lähenda n (algpiirid n) ,
  | otherwise
    = n >= 0
```

mis pöördub veel defineerimata muutujate `algpiirid` ja `lähenda` poole. Neist esimese rakendamine argumendile peab kätte andma sobiva otsingulõigu, millesse ruutjuur kindlasti jääb, teise rakendamine algsele argumendile ja alguslõigule leida lõigu poolitamise meetodil ülesande vastuse. Signatuurid on

```
algpiirid
  :: Integer -> (Integer , Integer)'

lähenda
  :: Integer -> (Integer , Integer) -> Bool'
```

Realiseerime lõigu poolitamise nii, et jooksva lõigu alumine otspunkt saab ruutjuurega ka võrdne olla, kuid ülemine otspunkt on ruutjuurest rangelt suurem. Kasutades deklaratsiooniga (37) antud operaatorit `sqr`, annab soovitava protsessi meile kood

```
lähenda n (a , b)
  | b - a <= 1
    = sqr a == n
  | otherwise
    = let
        m = (a + b) `div` 2
      in
      case compare (sqr m) n of '
        LT
          -> lähenda n (m , b)
        GT
          -> lähenda n (a , m)
        _
          -> True
```
(117)

Arvutus selle koodi järgi töötab üldjoontes nagu lõigu poolitamise meetod ülal kirjeldatud oli. Esimene valvur käsitleb lõike pikkusega 1, mispuhul tuleb vastus läbiproovimisega kindlaks teha, teine kõiki pikemaid lõike, kus tuleb tavaliselt poolitada. Esimeses juhus piisab proovida vaid lõigu alumist otspunkti, sest ülemine otspunkt pidi tingimuse kohaselt olema ruutjuurest suurem.

Igal rekursiivsel pöördumisel on tagatud, et uus otspunkt ei ole uuritava arvu täpne ruutjuur. Seega tingimus, et jooksva lõigu ülemine otspunkt ei ole täpne ruutjuur, tõepoolest säilib läbi arvutuse.

Jääb defineerida muutuja `algpiirid`. Algne lõik peab rahuldama kõiki tingimusi, millel algoritmi õigsus rajanes: ülemine otspunkt peab olema uuritava arvu ruutjuurest suurem, alumine aga ruutjuurest väiksem või võrdne temaga. Sobib näiteks definitsioon

```
algpiirid n
  = (1 , n).
```
(118)

Kuna alguslõiku arvutatakse reaalselt vaid 1-st suurema argumendiga, siis on argumendi ruutjuur tõepoolest argumendist väiksem ning 1 omakorda ruutjuurest väiksem.

Et alguslõik sisaldab algparameetri väärtusega n võrdse arvu täisarve, siis rekursiivsete pöördumiste arv on ligikaudu võrdne arvuga $\log n$. Kuna igal sammul tehakse vaid üks ruututõstmine, keskpunkti leidmine ja võrdlus, siis on operatsioonide arv kogu täisruuduks olemise kontrolli käigus võrdeline suurusega $\log n$.

Alguslõiku saab aga määrata kavalamalt, tehes n-ö pööratud kahendotsingut kogu 1-st paremale mineval poolsirgel. On ju selge, et n on üldiselt liiga jäme hinnang arvule $\sqrt{n}$. Selle asemel võib proovida järjest läbi arve, mis asuvad poolsirge algusest 1, 2, 4, 8 jne ühiku kaugusel, kuni esimese selliseni, mille ruut on etteantud arvust suurem. Alguslõigu otspunktideks saavad kaks viimast proovitud arvu; siis lõigu poolitamise realisatsioonis lõigule seatud tingimused ilmselt kehtivad. Kirjeldatud idee realiseerib kood

```
algpiirid n
  = let
      piirid x
        | sqr y > n
          = (x , y)
        | otherwise  .
          = piirid y
        where
          y = x + x
    in
    piirid 1
```

(Sarnasus kahendotsinguga ilmneb, kui vaadata protsessi läbi teisenduse, mis teeb kõik arvud pöördarvudeks.)

Selline alguslõigu leidmine nõuab ruututõstmisi ja võrdlusi leitud lõigu ülemise otspunkti logaritmi jagu. Et lähtuvalt realisatsioonist ei ületa ülemine otspunkt otsitavat ruutjuurt rohkem kui 2 korda, siis on nii selle otsingu kui ka järgneva lõigusisese otsingu (operaatoriga `lähenda`) töömaht võrdeline ruutjuure logaritmiga. Et aga arvu ruutjuure logaritm on võrdeline arvu enda logaritmiga (täpsemalt, on pool sellest), siis olulist ajavõitu me selle optimeeringuga siin ei saa.

Lõigu poolitamise meetodit saab mõistagi kasutada ka teiste kasvavate funktsioonide korral, samuti kahanevate korral. Viimatinähtud kavalus alguslõigu leidmisel on hea ettevaatusabinõu, kui funktsioon kasvab nii kiiresti, et umbropsu suure argumendiga proovimisel jääks arvutus võimatult aeglaseks.

Vaatleme veel samuti klassikalist ülesannet leida etteantud meelevaldses järjestuses esitatud andmete loetelust selline, mis on suurusjärjestuses etteantud numbriga kohal.

Haskellis programmeerimiseks täpsustame, et defineerida tuleb funktsioon, mis võtab argumendiks järjestusega tüüpi andmete listi l ja täisarvu n ning kui n on mittenegatiivne ja väiksem listi pikkusest, siis annab tulemuseks listi l elementide seas suuruselt n-ndal kohal oleva. (Nagu ikka, nummerdame elemente listis alates 0-st.) Signatuur võiks olla

```
suuruseltNr
  :: (Ord a)        .
  => Int -> [a] -> a
```

Sel ülesandel leidub “jaga ja valitse” strateegiaga lahendus, mille juhatab kätte kahendotsingu idee. Jaotame antud listi elemendid kahte lühemasse listi selliselt, et ühes on kõik elemendid väiksemad kui teises. Kui esimeses listis on elementide arv suurem kui otsitava objekti number suurusjärjestuses, siis jääb otsitav esimesse listi ja otsime sealt edasi samal viisil. Vastasel korral on otsitav teises listis, mispuhul otsime edasi teisest listist samal viisil.

Probleem on, kuidas selliseid kaheks jaotamisi saada. Kõige parem oleks muidugi, kui tekkivad osad sisaldaksid enam-vähem ühepalju elemente, sest siis kahaneks otsinguala igal rekursioonisammul umbes kaks korda nagu kahendotsingus. Et selline kaheks jaotamine on niisama keeruline kui ülesanne ise, siis loobutakse sellest eesmärgist ja tehakse selle asemel võimalikult lihtne jaotamine.

Võtame jaotamisel lahkmeks listi esimese elemendi (kõige lihtsamini kättesaadav) ja jaotame tema järgi listi ülejäänud elemendid. Lahkme järgi jaotamine on eespool defineeritud muutuja `kaheksLahuta` väärtuseks (kood (73) või (74) või (101)).

Vastavalt ülalesitatud ideedele saame siis soovitud operaatori definitsiooni

```
suuruseltNr n (x : xs)
  = let
      (us , vs)
        = kaheksLahuta x xs
      l = length us
    in
    case compare l n of
      GT                                          .      (119)
        -> suuruseltNr n us
      LT
        -> suuruseltNr (n - l - 1) vs
      _
        -> x
suuruseltNr _ _
  = error "suuruseltNr: mõõtmetest väljas"
```

Koodis toimub hargnemine kolmeks, sest listi saba kaheks lahutamisel jääb listi pea kummastki osast välja ja lisajuht (valikuavaldise kolmas) vastab sellele, kui otsitud kohal suurusjärjestuses on parajasti listi pea ehk lahe ise.

Isegi lahkme sellise valiku korral saavutatakse keskeltläbi ikka kahekordne otsinguala vähenemine rekursioonisammu kohta. Jaotamine etteantud lahkme järgi käib jaotatava osa suurusega võrdelise operatsioonide arvuga. Seega järjestikuste kaheksjagamiste töömahud on keskmisel juhul võrdelised arvudega l, $\frac{l}{2}$, $\frac{l}{4}$ jne, kus l on listi pikkus, ja nende summa on ligikaudu $l \cdot \left(1 + \frac{1}{2} + \frac{1}{4} + \ldots\right)$ ehk $2l$. Seega kogu arvutuse töömaht on ligikaudu võrdeline elementide arvuga listis sõltumata otsitava järjekorranumbrist.

Siiski on olemas võimalus, et kaheks jaotamised ei tekita ligilähedaselt võrdse pikkusega osi, siis võib arvutuseks kauem aega minna. Eriti halb juht on see, kui iga jaotamine elimineerib vaid ühe elemendi. Siis on töömaht kokku võrdeline andmete arvu ruuduga. Realisatsioonis definitsiooniga (119) juhtub see siis, kui listi elemendid on kasvavas järjestuses ja otsitakse suurimat, ning siis, kui listi elemendid on kahanevas järjestuses ja otsitakse vähimat. Mõlemal juhul jäävad igal rekursioonisammul kõik listi saba elemendid lahkmest ühele poole, eraldub vaid pea.

Ülesandeid

271. Defineerida muutuja `kaheAste` väärtuseks funktsioon, mis võtab argumendiks täisarvu n tüübist Integer ja annab välja tõeväärtuse vastavalt sellele, kas n on 2 aste või mitte.

272. Ülesandes 247 defineeriti kolmnurkarv. Defineerida muutuja `kolmnurkne` väärtuseks funktsioon, mis võtab argumendiks täisarvu n tüübist Integer ja annab välja tõeväärtuse vastavalt sellele, kas n on kolmnurkarv või mitte.

273. Defineerida muutuja `intSqrt` väärtuseks funktsioon, mis võtab argumendiks täisarvu n tüübist Integer ja kui see on mittenegatiivne, siis annab tulemuseks n ruutjuure täisosa. Negatiivse n puhul peab arvutus lõppema eestikeelse veateatega.

274. Anda muutuja `intLog` väärtuseks karritatud funktsioon, mis võtab argumentideks täisarvud a ja n ja kui $\log_a n$ eksisteerib, siis annab välja $\log_a n$ täisosa. Kui logaritmi ei leidu, peab arvutus lõppema eestikeelse veateatega.

275. Defineerida muutuja `cosFix` väärtuseks võrrandi $x = \cos x$ võimalikult täpne ujukomaarvuga väljendatud lähend.

276. Koodi (119) eeskujul defineerida operaator `kaheksSuuruselt`, mis võtaks samasugused argumendid nagu `suuruseltNr`, kuid mille väärtuseks olev funktsioon annaks argumentidel n ja l tulemuseks listipaari, kus listi l elementidest n tükki on esimeses komponentlistis ja kõik ülejäänud teises ning teise listi elemendid on kõik vähemalt niisama suured kui esimese omad. Kui n on negatiivne, siis olgu tulemus sama mis $n = 0$ korral; kui n on suurem l pikkusest, siis tulgu sama tulemus mis juhul, kui n on l pikkusega võrdne.

277. Kas muutuja `täisruut` väärtus muutub, kui definitsioonis (117) panna esimese valvuri juhu parema poole avaldise `b - a <= 1` asemele `False`?

2. Mitme mittetriviaalse alamülesandega olukorrad. Siin pangem tähele, et kui ülesanne seisneb listi ehitamises ja alamülesannete tulemused tuleks konkateneerida, siis tasuks list ehitada akumulaatoris. Kuid esimese töötava koodi võib realiseerida ka niisama; arvutuse üleviimine akumulaatoritehnikale on tehtav lihtsa, mehhaanilise koodimuutmisega.

Vaatame nüüd tuntud Hanoi tornide ülesannet, mida imperatiivse programmeerimise õpetamisel kasutatakse tihti näitena rekursiooni kasulikkusest. Lahendus on aga just “jaga ja valitse” tüüpi algoritm.

Hanoi tornide ülesanne seisneb järgmises. On antud n erineva läbimõõduga ketast auguga keskel ja kolm püstist varrast, kuhu saab kettaid peale panna. Alguses on kõik kettad esimesel vardal alt üles suuruse kahanemise järjekorras. Ühe sammuga on lubatud tõsta suvaliselt vardalt väikseim ketas mõnele teisele vardale, kus pole temast väiksemaid kettaid. Kuidas võimalikult väikese arvu sammudega saavutada olukord, kus kõik kettad on teise varda peal?

Lahenduse võti peitub tähelepanekus, et suurima ketta ümbertõstmiseks peavad kõik teised olema enne ümber paigutatud, kusjuures nad kõik peavad olema kolmanda varda otsas, sest suurimat ketast saab asetada ainult tühja varda otsa. Pärast suurima ketta nihutamist suurim ketas enam rolli ei mängi, teised saab tema peale asetada niisama hästi kui tühjale vardale.

Seega ülesanne jaguneb järgmisteks alamülesanneteks: $n-1$ väiksema ketta ümbertõstmine esimeselt vardalt kolmandale; suurima ketta tõstmine esimeselt vardalt teisele; $n-1$ väiksema ketta tõstmine kolmandalt vardalt teisele. Seejuures äärmised alamülesanded on analoogilised tervikuga, keskmine aga triviaalne.

Defineerime muutuja `hanoi` väärtuseks funktsiooni, mis võtab argumendiks ketaste arvu n ja annab tulemuseks ümbertõstmiste jada, mis n ketta korral viib sihile. Loeme varraste nimedeks sümbolid A, B, C ja märgime üht ümbertõstmist kahetähelise sõnega, mille esimene täht näitab varrast, millelt väikseim ketas tuleb võtta, ning teine täht varrast, kuhu ketas tuleb panna. Siis sobib tüübisignatuur

```
hanoi
  :: Int -> [String].
```

Kuna lähtevarras, sihtvarras ja abivarras on erinevates alamülesannetes erinevad, tuleb appi võtta lokaalne operaator, mis need vardad lisaargumentides ette saab. Esimese lähendusena võib kirjutada koodi

```
hanoi n
  | n >= 0
    = let
        hanoi 0 _ _ _
          = []
        hanoi n x y z
          = let
              r = n - 1                                    ,
            in
            hanoi r x z y ++ [x, y] : hanoi r z y x
      in
      hanoi n 'A' 'B' 'C'
  | otherwise
    = error "hanoi: negatiivne argument"
```

(120)

kus `x`, `y`, `z` tähendavad vastavalt lähtevarrast, sihtvarrast ja abivarrast. Abioperaatori definitsioon seega ütleb, et selleks, et tõsta n ketast vardalt x vardale y varda z kaudu, kus $n > 0$, tuleb kõigepealt tõsta $n-1$ ketast vardalt x vardale z varda y kaudu, siis teha tõste vardalt x vardale y ja lõpuks tõsta $n-1$ ketast vardalt z vardale y varda x kaudu, 0 ketta tõstmiseks aga pole vaja ühtki tõstet teha.

Väärtustades näiteks `hanoi 2`, on tulemuseks list elementidega “AC”, “AB”, “CB”, mis ütleb, et 2 ketta korral tuleb tõsta algul ketas esimeselt vardalt kolmandale, siis ketas esimeselt vardalt teisele ja lõpuks kolmandalt teisele.

Kahjuks esineb koodis (120) rekursiivne pöördumine operaatori `++` vasakus argumendis. See tähendab, et tekib korduv kopeerimine, täpselt nagu listi ümberpööramise naiivse realisatsiooni (99) puhul. Esimesel rekursioonitasemel kopeeritakse pool kogu loodavast listist, järgmisel rekursioonitasemel kummagi pöördumise järgi kopeeritakse omakorda pool sellest poolest ehk kokku jällegi pool kogu listist, kolmandal sügavustasemel analoogiliselt ka pool kogu listist jne.

Olukorra parandamiseks piisab lisada lokaalsesse operaatorisse akumulaator ja mudida definitsiooni, et tulemuslist koostataks selles. Sobib kood

```
hanoi n
  | n >= 0
    = let
        hanoi 0 _ _ _ as
          = as
        hanoi n x y z as
          = let
              r = n - 1
            in
            hanoi r x z y ([x, y] : hanoi r z y x as)
      in
      hanoi n 'A' 'B' 'C' []
  | otherwise
    = error "hanoi: negatiivne argument"
```

(121)

Selle järgi arvutades kirjutab iga abioperaatori rakendamine oma tõstete jada akumulaatori algusse; kui akumulaatoris juba on midagi ees, jääb see puutumatult juurdekirjutatava osa järele. Algsel väljakutsel on akumulaator tühi, nii et lõpptulemus on sama mis koodi (120) korral, kuid nüüd satub iga element kohe oma lõplikku asupaika, tühiümbertõstmisi ei toimu.

Vaatame veel ülesannet leida järjestusega tüüpi elementide listi järgi list, kus samad elemendid oleksid mittekahanevalt järjestatud. Klassikalised “jaga ja valitse” stiilis lahendusmeetodid on kiirmeetod ja põimemeetod; realiseerime siin kiirmeetodi. Signatuur võiks olla

```
järjestaKiir
  :: (Ord a)
  => [a] -> [a]
```

Kiirmeetodi idee on sama mis suurusjärjestuses kindlal kohal paikneva elemendi otsimisel, mida eespool vaatlesime. Nagu sealgi, jaotatakse listi elemendid kahte lühemasse listi nii, et ühes listis on kõik elemendid väiksemad teise listi kõigist elementidest. Kumbki list järjestatakse samal meetodil. Seejärel pole lõpptulemuse saamiseks vaja muud teha kui need järjestatud listid üksteise järele kirjutada.

Seega lahenduse esimene, "jagamise" etapp on sama ja võib kasutada koodiga (73) või (74) või (101) defineeritud operaatorit `kaheksLahuta`. Erinevused on "valitsemise" osas ning triviaalse juhu — tühja listi — käsitlemises.

Nagu Hanoi tornide ülesandes, on siingi vaja alamülesannete lahenduseks olevate listide elemendid järjest ühte listi kirjutada. Teame juba, et siis on kasulik listi akumulaatoris koostada. Nii saame definitsiooni

```
järjestaKiir xs
  = let
      kiir (x : xs) as
        = let
            (us , vs)
              = kaheksLahuta x xs
          in
            kiir us (x : kiir vs as)
      kiir _        as
        = as
    in
    kiir xs []
```
, (122)

kus lokaalse operaatori argument `as` on mainitud akumulaator.

Et kogu jagamise etapp on ühine suurusjärjestuses teatud kohal olevate elementide otsimise ülesande lahendusega, on ka järjestamise kiirmeetodi töömaht kapriisne ja sõltub sellest, kui ligilähedaselt võrdseteks osadeks jaotamine toimub. Head ja halvad juhud on tolle ülesandega ühised. Keskmisel ja parematel juhtudel on kiirmeetodi töökulu laias laastus võrdeline suurusega $n \log n$, kus n on listi pikkus. Tööd tehakse rohkem kui suurusjärjestuses üksikute elementide otsingul, sest mittetriviaalseid alamülesandeid on ühe asemel kaks. Halvimal juhul, kui igal rekursioonisammul eraldub ainult lahe, tuleb siingi ruutsõltuvus.

Märgime, et suurusjärjestuses kindlal kohal olevate elementide otsimiseks suhteliselt halvasti loetava koodiga (119) defineeritud operaatorile `suuruseltNr` võib nüüd anda hoopis lihtsama samaväärse definitsiooni

```
suuruseltNr n xs
  | 0 <= n && n < length xs
    = järjestaKiir xs !! n
  | otherwise
    = error "suuruseltNr: mõõtmetest väljas"
```
, (123)

mis lihtsalt laseb listi järjestada ja võtta tulemusest nõutud positsioonilt elemendi.

Kõige huvitavam on seejuures, et kui tarvis läheb vaid suurusjärjestuses üsna ees asetsevaid elemente, st arvuline argument on ülalt tõkestatud, on definitsioon (123) niisama efektiivne kui kood (119).

See on järjekordne laisa väärtustamise paradoks. Seda on raske konkreetselt põhjendada, aga katsetamine seda näitab: kuigi järjestamine on keerukam protsess kui järjestuses kindla elemendi otsimine, tehakse definitsiooni (123) korral tööd ikkagi võrdeliselt ühe elemendi otsimise protsessi töömahuga.

Ülesandeid

278. Bababababi keeles on kasutusel ainult tähed A ja B. Sõnade moodustamisel kehtivad järgmised ranged reeglid.

    1. A on sõna.
    2. Kui u ja v on ühepikkused bababababi sõnad, siis $u + v'$ ja $u + v^*$ on bababababi sõnad, kus
        - w' tähistab sõna, mille saame sõnast w tähtede järjekorra vastupidiseks muutmisel,
        - w^* on sõna, mille saame sõnast w iga tähe väljavahetamisel vastandtähega (st A-de asendamisel B-de ja B-de asendamisel A-dega),
        - $u+v$ tähistab sõna, mille saame, kui paarituarvulistele positsioonidele paneme järjekorras u tähed ja paarisarvulistele järjekorras v tähed (muutuja `vaheliti` väärtus ülesandest 210).
    3. Kõik sõnad on saadavad punktide 1 ja 2 abil.

    Defineerida muutuja `bababababi` väärtuseks funktsioon, mis võtab argumendiks sõne ja annab tulemuseks `True` või `False` vastavalt sellele, kas ta on bababababi keele sõna või ei.

279. Realiseerida järjestamise kiirmeetodi teisend, mille korral järjestamise käigus ühtlasi kaotatakse kordumised, st tulemuslistis esinevad kasvavas järjestuses kõik algse listi elemendid, igaüks üks kord.

280. Realiseerida järjestamise kiirmeetodi teisend, mille korral tulemusena tekib paaride list, kus paaride esimesed komponendid on parajasti argumentlistis esinevad elemendid, igaüks ühe korra, kasvavas järjestuses ning vastavad teised komponendid näitavad elemendi esinemise kordsust argumentlistis.

281. Kooskõlas signatuuriga

```
kordusteta
  :: (Ord a)
  => [a] -> Bool
```

defineerida "jaga ja valitse" strateegia abil muutuja `kordusteta` väärtuseks funktsioon, mis annab lõplikul listil välja True, kui listis ükski element ei kordu, ja False vastasel korral.

282. **Permutatsioon** objektidest a_1 kuni a_n on järjend, mille komponendid on needsamad objektid mingis järjekorras (igaüks esineb täpselt ühe korra).

Kooskõlas signatuuriga

```
onPerm
  :: [Int] -> Bool
```

defineerida muutuja `onPerm` väärtuseks predikaat, mis kontrollib etteantud listi kohta, kas on tegemist permutatsiooniga täisarvudest 1 kuni mingi n.

3. Alamülesannete listid. "Jaga ja valitse" strateegia traditsioonilises realisatsioonis vastab igale alamülesandele üks rekursiivne pöördumine arvutuse käigus, mis seda ülesannet lahendab.

Kõigi seni vaadeldud näidete puhul on see nii.

Funktsionaalse keele väljendusrikkus teeb võimalikuks ka teistsugused realisatsioonid. Üks elegantne lähenemine on koostada triviaalsetest alamülesannete lahendustest list ja tasehaaval neid kombineerida suuremate alamülesannete lahendusteks, kuni lõpuks jõutakse algse ülesande lahenduseni.

Omavahel kombineeritavad alamülesanded võivad olla täpselt samad mis traditsioonilises tehnikas, nii et arvutuse kulg on põhimõtteliselt samasugune, erinevused on vormilised. Siiski on mõnikord võimalik teistsugune jaotus alamülesanneteks, millega võidetakse töökiiruses.

Realiseerime näitena järjestamise põimemeetodi, tehes seda algul traditsiooniliselt ja siis alamülesannete listiga. Signatuur on igal juhul

```
järjestaPõim
  :: (Ord a)   .
  => [a] -> [a]
```

Erinevalt kiirmeetodist, on põimemeetodil järjestamise põhimõtteks jagada list kärmelt kaheks võimalikult ühepikkuseks osaks, kasutamata ühtki võrdlemist, järjestada kumbki osa samal meetodil ning põimida tulemused üheks järjestatud listiks.

Kahe järjestatud listi põimimisega oleme juba kokku puutunud, seda realiseeris definitsioon (91). Seal oli vaja ühesuguste elementide kordumised kaotada, järjestamisülesandes aga tavaliselt tahetakse kõik koopiad alles hoida. Seepärast anname siin muutujale `põim` uue definitsiooni

```
põim xs@ (a : as) ys@ (b : bs)
  | a <= b
    = a : põim as ys
  | otherwise
    = b : põim xs bs          ,                    (124)
põim xs           []
  = xs
põim _            ys
  = ys
```

mille järgi arvutamisel kordusi ei kaotata.

Nüüd saaksime põimemeetodil järjestamise defineerida näiteks koodiga

```
järjestaPõim xs@ (_ : _ : _)
  = let
      (us , vs)
        = kaheksLoe xs
    in                                                .  (125)
    põim (järjestaPõim us) (järjestaPõim vs)
järjestaPõim xs
  = xs
```

Listi jagamiseks kaheks võimalikult ühepikkuseks osaks on kasutatud muutujat `kaheksLoe`, mis võib olla defineeritud koodiga (76), (77), (80) või (109).

Esimene deklaratsioon definitsioonis (125) sobib juhul, kui listis on vähemalt kaks elementi. Sellisel juhul on mõlemad kahekslugemisega saadavad osad algsest listist lühemad, nii et ülesande suurus rekursiivsel pöördumisel väheneb. Kui listis on vähem kui kaks elementi, siis sobib teine deklaratsioon ja antakse originaallist välja. See on korrektne, sest 0- ja 1-elemendilised listid on juba järjestatud.

Põimemeetodil järjestamine definitsiooniga (125) sooritab alati laias laastus suurusega $n \log n$ võrdeliselt samme, kus n on listi pikkus, sest jaotamine teeb listi peaaegu täpselt pooleks sõltumata elementide paiknemisest listis. Kui aga list on juba järjestatud või vaid mõned elemendid on vales positsioonis, on ka see liiga ebaefektiivne, sest siis saaks tulemuse leida pikkuse suhtes lineaarse tööajaga.

Kirjutame nüüd definitsiooni alamülesannete listi abil. Kaheksjagamise protsessi võib lihtsalt ära jätta, sest seal midagi sisulist ei toimu. Lähtume jagamisprotsessi lõpptulemusest, milleks on algse listi hakitus triviaalseteks järjestatud listijuppideks; paneme kõik need listijupid ühte (n-ö alamülesannete) listi. Seejärel hakkame neid listijuppe kahekaupa põimima, nii et tekivad järjest pikemad järjestatud listid, kuni lõpuks on kõik elemendid ühes järjestatud listis.

Definitsiooni (125) korral toimuvat jagamist järgides võime juppideks võtta üksikutest elementidest moodustatud listid, sest kõigi pikemate listide puhul jagatakse edasi. On lihtne defineerida funktsioon, mis algse listi järgi sellise juppide listi annab; sobib kood

```
jupid
  :: (Ord a)     ,
  => [a] -> [[a]]

jupid xs
  = [[x] | x <- xs]
```
. (126)

(Signatuur võiks ka üldisem olla, kuid lähtume ülesande kontekstist.) Üheelemendilised jupid moodustavad hargnemiste puu kõige alumise taseme ehk lehetaseme.

Kahekaupa põimimise programmeerimisel peab jälgima, et põimimiste skeem tuleks kahendpuukujuline. Selleks tuleb põimida tasehaaval.

Ühe taseme põimimiseks defineerime muutuja `põimiTase` koodiga

```
põimiTase (xs : ys : yss)
  = põim xs ys : põimiTase yss
põimiTase xss
  = xss
```
, (127)

signatuur on

```
põimiTase
  :: (Ord a)       .
  => [[a]] -> [[a]]
```

Operaatori `põimiTase` ühe rakendamise tulemusel jääb elementlistide arv umbes kaks korda väiksemaks, kuid nad sisaldavad jätkuvalt kõik algse listi elemendid.

Selliseid põimimisi on vaja korrata, kuni elementlistide arv kahaneb 1-ni. Nii tekib puukujuline arvutus. Lõpuks on vaja see ainus list välja anda. Kõige selle tarvis on signatuur ja definitsioon

```
põimiPuu
  :: (Ord a)     ,
  => [[a]] -> [a]
```

```
põimiPuu xss@ (_ : _ : _)
  = põimiPuu (põimiTase xss)
põimiPuu [xs]
  = xs
```
(128)

Itereerimine toimub akumulaatoris. Tühja listi jaoks deklaratsiooni pole esitatud, kuna kanname hoolt, et seda operaatorit tühjal listil kunagi välja ei kutsutaks.

Operaatori `järjestaPõim` uueks definitsiooniks on

```
järjestaPõim []
  = []
järjestaPõim xs
  = põimiPuu (jupid xs)
```

Kuna saadud järjestamisprotsess toimub sisuliselt samamoodi kui eelmine, on töömaht endiselt võrdeline suurusega $n \log n$, kus n on listi pikkus, sõltumata listist.

Kuid nüüd järgneb puänt. Ilmselt on algse listi jupitamise juures olulised parajasti kaks asja: et jupid sisaldaksid kokku parajasti algse listi kõik elemendid ja et iga jupp oleks järjestatud (viimane on vajalik, sest muidu ei tööta põimimine korrektselt). Pole oluline, et põimimise etapp algaks üheelemendilistest listidest. Me võime juppideks niisama hästi võtta algse listi juba järjestatud lõigud ehk segmendid.

Siis lõpetab järjestamisprotsess seda kiiremini, mida rohkem elemente algses listis on juba järjestatud. Kui algne list on üleni järjestatud, siis on kogu järjestamisprotsess triviaalne ja töötab listi pikkusega võrdelise ajaga (ainsa segmendi leidmiseks tuleb list korra läbi vaadata).

Seega meil on vaja vaid operatsiooni, mis leiaks listi järgi tema segmentide listi. Kuid see on meil juba olemas, definitsioonidega (78) ja (79) muutuja `segmendid` väärtuseks antud. Niisiis piisab kood (126) asendada koodiga

```
jupid xs
  = segmendid xs
```
(129)

Kõigele krooniks märgime, et kui nüüd operaatori `suuruseltNr` definitsioonis (123), mis sätestas listis suuruselt kindlal kohal oleva elemendi leidmise selle listi järjestamise ja siis positsiooni järgi lugemise kaudu ja mis kummalisel kombel töötab üldiselt efektiivselt, asendada kiirmeetod põimemeetodiga, siis saame veel kiirema arvutuse, mida ka ükski halb juht ei kummita.

Ülesandeid

283. Realiseerida alamülesannete listiga järjestamise põimemeetodi teisend, mille korral järjestamise käigus ühtlasi kaotatakse kordumised, st tulemuslistis esinevad kasvavas järjestuses kõik algse listi elemendid, igaüks ühe korra.

284. Realiseerida alamülesannete listiga järjestamise põimemeetodi teisend, mille korral tulemusena tekib paaride list, kus paaride esimesed komponendid on parajasti argumentlistis esinevad elemendid, igaüks ühe korra, kasvavas järjestuses ning vastavad teised komponendid näitavad elemendi esinemise kordsust argumentlistis.

285. Kujutame ette olümpiasüsteemis (st kaotajad langevad välja) turniiri, milles võistlevad listid ja milles mäng kahe listi vahel seisneb listide "jooksvate" elementide võrdlemises. Mängu võidab see list, kelle jooksev element on suurem. Kui jooksvad elemendid on võrdsed, siis mängivad samad listid omavahel kohe järgmiste elementidega uuesti, kuni tulemus selgub (võib eeldada, et kokku ei satu kaks võrdset lõpmatut listi). Kui ühel listil saavad elemendid otsa, on ta kaotanud; kui otsa saavad mõlemad listid, on mõlemad kaotanud.

    Defineerida muutuja `listiOlümpia` väärtuseks funktsioon, mis võtab argumendiks võrreldavate elementidega listide listi ja annab tulemuseks Just l, kus l on turniiri võitjalist, kui võitja selgub, ja Nothing, kui võitjat ei selgu. Turniiri esimeses voorus alustab iga list mängu esimese elemendiga; igas järgmises voorus tuleb võtta uus jooksev element, v.a juhul, kui viimane mäng jäi vastase puudumisel või tema elementide otsalõppemise tõttu ära.

286. Lahendada ülesande 285 modifikatsioon, kus elementide otsalõppemisel võetakse jooksvaks elemendiks jälle esimene. Kuna võitja selgub alati, peaks kirjeldatava funktsiooni väärtuseks olema võitjalist l, mitte Just l.

4.3.2 Dünaamiline programmeerimine

Mõnikord on vahetu "jaga ja valitse" realisatsioon ebaefektiivne, sest alamülesanded hakkavad korduma, kuid seda ei kasutata ära.

Üks näide ülesandest, kus "jaga ja valitse" strateegia on juba püstituses sisse kodeeritud, on Fibonacci arvude leidmine järjekorranumbri järgi. Jada matemaatiline spetsifikatsioon (66) näitab, et see ülesanne parameetri väärtuse n korral taandub kahele sarnasele alamülesandele vastavalt parameetrite väärtustega $n-1$ ja $n-2$. Otse selle spetsifikatsiooni järgi kirjutatud kood (67) oli seega meie esimene näide "jaga ja valitse" tüüpi mõtlemisest.

Nägime aga kohe, et kood (67) on praktiliseks kasutamiseks kõlbmatu, sest arvutuse maht kasvab eksponentsiaalselt, põhjuseks alamülesannete korduv lahendamine.

Programmeerimise võtet, kus alamülesannete lahenduste tulemusi kasutatakse korduvalt ilma neid uuesti lahendamata, nimetatakse dünaamiliseks programmeerimiseks. Tulemuseks võib olla kood, kust algset (või vahepealset) "jaga ja valitse" stiilis püstitust on lausa raske ära tunda.

Ülal vaadeldud Fibonacci arvude arvutamise optimeerimised olid kõik dünaamilise programmeerimise näited.

1. Üksikute vahetulemuste meeldejätmine. Lihtsamal juhul piisab korduvatest arvutustest lahtisaamiseks ühe või paari andme meeleshoidmisest igal rekursioonisammul.

Näiteks Fibonacci arvude arvutamisel võis kasutada abifunktsiooni, mis annab üksikute arvude asemel välja arvupaare (kood (68)), või kahe lisaargumendiga abifunktsiooni (kood (106)).

Vaatame, kuidas võiks programmeerida arvu astendamise täisarvuga, mis positiivsel astendajal defineeritakse korduva korrutamise kaudu, negatiivsel aga vastandarvulise astme pöördarvuna. Signatuur oleks

```
aste
  :: (Fractional a, Integral b).
  => a -> b -> a
```

Kandes matemaatilise definitsiooni naiivselt üle Haskelli, saaksime koodi

```
aste a n
  = case compare n 0 of
      GT
        -> a * aste a (n - 1)
      EQ
        -> 1
      _
        -> 1 / aste a (-n)
```
, (130)

mille järgi arvutades on töömaht võrdeline astendajaga.

Efektiivsema arvutuse saamiseks võtame appi “jaga ja valitse” strateegia: tekitada kaks võrdse suurusega mõttelist tegurite rühma, arvutada kummagi rühma tegurite korrutis ja leida nende põhjal lõpptulemus.

Kui astendaja on paaris, siis saab kõik tegurid kahte ühesuurusse rühma ära jagada. Kuna korrutis on sama, sest tegurid on samad ja neid on ühepalju, siis piisab arvutada ühe rühma tegurite korrutis ja see ruutu tõsta.

Kui astendaja on paaritu, siis korduvkasutatavuse tekitamiseks jätame ühe teguri rühmadest välja. Siis on kaks rühma jälle ühesugused, piisab leida neist ühe tegurite korrutis, ruutu tõsta ning tulemus algul eraldi pandud teguriga läbi korrutada.

Nii saame igal rekursioonisammul läbi ühe rekursiivse pöördumisega. Kui astendaja on 0, siis on tegu triviaalse juhuga, kus anname kohe lõppvastuse 1 välja. Negatiivse

astendaja juhu käsitleme eraldi. Kokku tuleb kood

```
aste a n
  = case compare n 0 of
      GT
        -> let
             (q , r)
               = n `divMod` 2
             x = aste a q                          . (131)
           in
           if r == 0 then x * x else a * x * x
      EQ
        -> 1

      _
        -> 1 / aste a (-n)
```

Arvutades selle koodi järgi, väheneb astendaja igal rekursiivsel pöördumisel ligikaudu 2 korda, nii et operatsioonide koguarv on võrdeline astendaja logaritmiga.

Tegureid teistmoodi grupeerides võiksime saada teisi realisatsioone, mis annavad logaritmilise töömahuga arvutuse, näiteks

```
aste a n
  = case compare n 0 of
      GT
        -> let
             (q , r)
               = n `divMod` 2
             y = aste (a * a) q          .          (132)
           in
           if r == 0 then y else a * y
      EQ
        -> 1

      _
        -> 1 / aste a (-n)
```

Võtmesammuks efektiivsuse kolossaalse paranemise juurde on siin mitte “jaga ja valitse” strateegia iseenesest, vaid dünaamiline programmeerimine, millega saavutatakse kahe rekursiivse pöördumise asemele üks. Kui näiteks koodis (131) lokaalset muutujat `x` mitte defineerida ja kasutada `x` asemel avaldist `aste a q`, oleks kogu ümbertöötamise vaev kasutu, korrutamiste arv võrduks vanaviisi astendajaga.

Pöördume veel tagasi ka Hanoi tornide ülesande juurde. Oletame, et eesmärgiks on leida mitte sihile viiv tõstete jada, vaid selle käigus esinevate seisude jada. Esitame seisu listikolmikuna, kus kolmiku komponendid vastavad varrastele ja iga list

koosneb parajasti vastava varda otsas olevate ketaste järjekorranumbritest. Kettaid nummerdame suuruse kasvamise järjestuses.

Kodeerimist võib alustada tüübisünonüümideklaratsioonist

```
type Seis
  = ([Int] , [Int] , [Int])
```

ja sellega sisse toodud tüübisünonüümi abil kirjutatud signatuurist

```
hanoiSeisud
  :: Int -> [Seis]
```

Jagame ülesande osadeks nii nagu enne. Selleks, et kahe rekursiivse pöördumise asemel ühega hakkama saada, tuleb loobuda vardaparameetritest ning selle asemel rekursiivse pöördumise tulemust kahte moodi tõlgendada. Tõlgendamine seisneb varraste ümbermängimises juba väljaarvutatud seisudes — põhimõtteliselt sama asi mis varasemate koodide (120) ja (121) puhul toimus argumentide vahetamisega.

Kood

```
hanoiSeisud n
  = case compare n 0 of
      GT
        -> let
             hs
               = hanoiSeisud (n - 1)
           in
           [(n : xs , zs , ys) | (xs , ys , zs) <- hs] ++
           [(zs , n : ys , xs) | (xs , ys , zs) <- hs]
      EQ
        -> [([] , [] , [])]
      _
        -> error "hanoiSeisud: neg. argument"
```

(133)

need mõtted ka realiseerib. Rekursiivse pöördumise tulemus on kasutatud kahel viisil: loodava listi esimeses pooles vahetatakse igas seisus ära teine ja kolmas varras ning lisatakse kõikjal esimesele vardale suurim ketas, sarnaselt on teises pooles vahetuses esimene ja kolmas varras ja suurim ketas on seal lisatud teisele vardale.

Ülesandeid

287. Anda muutuja `linfunAste` väärtuseks funktsioon, mis võtab sisse murdarvupaari (a, b) ja täisarvu k ning annab välja sellise paari (c, d), et funktsioon $h(x) = cx + d$ võrdub funktsiooniga $f(x) = ax + b$ astmes k (s.o positiivse k

korral k-kordselt järjest rakendatud f, muidu f^{-k} pöördfunktsioon). Kui niisugust astet ei leidu ($a = 0$ ja $k < 0$), siis peab arvutus lõppema veateatega. Operatsioonide arv olgu võrdeline astendaja absoluutväärtuse logaritmiga.

2. Meeldejätmine listis. Keerulisemal juhul tuleb korraga meeles hoida teadmata arvul alamülesannete vastuseid. Olenevalt ülesandest sobib need organiseerida listi, listide listi või keerulisemasse struktuuri. Laisa väärtustamise tõttu on kõige lihtsamaks-loomulikumaks valikuks lõpmatud listid.

Fibonacci arvude arvutamiseks kirjutatud koodist kasutas listi definitsioon (93), list oli defineeritud koodiga (92).

Listist lugemisega lahendus on võimalik ka astendamise ülesandes. See tähendaks mitte kõigi astmete, vaid ainult järjestikuste ruututõstmiste teel saadavate astmete salvestamist listi. Lähtudes koodist (132), võiksime saada definitsiooni

```
aste a n
  | n >= 0
    = let
        as
          = a : [a * a | a <- as]
        aste 0 _
          = 1
        aste n (a : as)
          = let
              (q , r)
                = n `divMod` 2
              y = aste q as
            in
            if r == 0 then y else a * y
      in
      aste n as
  | otherwise
    = 1 / aste a (-n)
```

See kood töötab vahestruktuuri tõttu pisut aeglasemalt.

Veel ühe näitena võtame Catalani jada

$$1 \quad 1 \quad 2 \quad 5 \quad 14 \quad 42 \quad 132 \quad 429 \quad 1430 \quad 4862 \; 16796 \; 58786 \; \ldots,$$

mis matemaatikas defineeritakse rekurrentselt tingimustega

$$c_0 = 1, \qquad \forall n \geqslant 1 \left(c_n = \sum_{i=0}^{n-1} c_i c_{n-1-i} \right). \tag{134}$$

Catalani jada liikmeid nimetatakse Catalani arvudeks.

Seoste (134) otsetõlge Haskelli annaks ilmselt väga ebaefektiivse arvutuse, sest sõltumatuid rekursiivseid pöördumisi oleks väga palju. Seetõttu on otstarbekam defineerida terve Catalani jada muutuja `cats` väärtuseks vastavalt signatuurile

```
cats
  :: [Integer]
```

Catalani jada kui terviku kirjeldamiseks paneme tähele, et korrutiste summa rekurrentse seose paremal pool sarnaneb sellega, kuidas avaldub kindla astme kordaja kahe polünoomi korrutises tegurpolünoomide kordajate kaudu. Et korrutisena saada kindlat astet, tuleb kõikvõimalikel viisidel valida üks liige ühes ja teine teises tegurpolünoomis, nii et nende astmete summa tuleks vajalik arv, ja kordaja kujuneb selliste liikmete kordajate korrutiste summana. Näiteks x^2 kordaja polünoomis $(2+3x+x^2)\cdot(5+4x+6x^2-x^3)$ on $2\cdot 6+3\cdot 4+1\cdot 5$ ehk 29. Catalani arvu avaldises võetakse mõlemad tegurid samast, Catalani jadast. Seega Catalani jada on justkui sellise lõpmatu polünoomi $c(x)$ kordajate jada, mille ruudu (iseendaga korrutise) kordajate jada on sama jada ilma esimese liikmeta (sümbolites $c(x) = 1 + c^2(x)\cdot x$).

Seepärast realiseerime kõigepealt polünoomide korrutamise, kus polünoomid on esitatud listidena kordajatest astmete kasvamise järjekorras (astmed vastavad täpselt indeksile listis, sest mõlemad algavad 0-st); niisugust operatsiooni nimetatakse konvolutsiooniks. Olgu tema signatuur

```
(*#*)
  :: (Num a)          .
  => [a] -> [a] -> [a]
```

Muidugi pole konvolutsiooni kirjeldamisel Haskellis mõistlik viidata listi üksikutele elementidele operaatori `!!` abil. Parem on mittetühjade kordajate jadade korral eraldada ühes polünoomis vabaliige, korrutada teise polünoomiga see ja rekursiivselt ka ülejäänud osa ning liita tulemused. Tühja kordajate jadaga polünoom on konstantselt null ja sellega korrutamine annab samuti tühja kordajate jada.

Seega on vaja polünoomide liitmist, mis seisneb vastavate kordajate liitmises. Meie esituse puhul sobivad signatuur ja definitsioon

```
liidaVastavad
  :: (Num a)          ,
  => [a] -> [a] -> [a]

liidaVastavad (a : as) (b : bs)
  = a + b : liidaVastavad as bs
liidaVastavad xs       []
  = xs                          .
liidaVastavad _        ys
  = ys
```
(135)

Konvolutsiooni saame nüüd deklaratsiooniga

```
xs@ ~(a : as) *#* ys@ ~(b : bs)
  | null xs || null ys
    = []
  | otherwise
    = a * b :
      liidaVastavad [a * y | y <- bs] (as *#* ys)
```
, (136)

kus vabaliige on eraldi välja toodud selleks, et liitmisel vastavad astmed kohakuti satuksid.

Lõpuks on soovitud muutuja `cats` juba lihtsasti defineeritav koodiga

```
cats
  = 1 : cats *#* cats.
```
(137)

Võib märkida, et tegelikult on Catalani arvud avaldatavad binoomkordajate kaudu valemiga $c_n = \frac{C_{2n}^n}{n+1}$, mis võimaldab neid veelgi efektiivsemalt arvutada.

Dünaamilise programmeerimise poole tuleb vaadata ka kahe argumendiga funktsiooni kodeerimisel, kui tema väärtus suvalisel mittetriviaalsel argumendipaaril on esitatud tema väärtuste kaudu mitmel väiksemate komponentidega argumendil. Siingi tekitaks rekurrentse võrrandi otsetõlge Haskelli palju korduvaid alamülesandeid. Selle asemel tasub kirjeldada korraga terve funktsiooni väärtuste tabel, seejuures vältides indeksi järgi otsimist, mis lisaks tarbetut arvutusmahtu.

Analoogiline lähenemine töötab ka suurema argumentide arvu korral.

Binoomkordajate C_n^k arvutamisel tähendab kirjeldatud üleminek, et kirjeldatakse korraga terve Pascali kolmnurk. Selle arvutamiseks sobib tuntud nn **Pascali reegel** $C_n^k = C_{n-1}^{k-1} + C_{n-1}^k$, mis kehtib juhul $0 < k < n$. Juhtudel $k = 0$ ja $k = n$ võrdub binoomkordaja 1-ga.

Ehkki binoomkordajad saab avaldada faktoriaali või kahaneva faktoriaali kaudu ning üksikute binoomkordajate leidmisel on see Pascali reegli abil arvutusest tunduvalt efektiivsem, kirjeldame siin näitena Pascali kolmnurga listide listi kujul, sest see on lugejale arvatavasti kõige tuttavam sedasorti tabel.

Realiseerime kõigepealt Pascali kolmnurga tavaesituse ridade kaupa; signatuur olgu

```
pasRead
  :: [[Integer]].
```

Selleks on vaja osata arvutada jooksva rea järgi uus rida, kirjutades otstesse ühed ja vahele jooksva rea naaberarvude summad. Sarnane funktsioon oli koodi (75) abil seotud muutujaga `kahesummad`, kuid see jätab ühed algusse ja lõppu panemata.

Kood, mis selle puuduse kõrvaldab, lisades summade listi algusse etteantud listi esimese ja lõppu tema viimase elemendi, oleks

```
kahesummad' (x : xs)
  = let
      summad x (y : ys)
        = x + y : summad y ys
      summad x _
        = [x]
    in
    x : summad x xs
```
. (138)

(Tühja argumentlisti juht on käsitlemata, kuna seda ei tule Pascali kolmnurga arvutamisel ette ja seda pole ka võimalik kuidagi mõistlikult üldiselt sätestada.)

Pascali kolmnurga ridade listi kirjeldab nüüd definitsioon

```
pasRead
  = let
      bss
        = [1] : [kahesummad' bs | bs <- bss]
    in
    bss
```
. (139)

Üksikute binoomkordajate arvutamiseks võib lisada koodi

```
bin
  :: Int -> Int -> Integer'

bin n k
  | n >= 0 && k >= 0
    = if n >= k then pasRead !! n !! k else 0
  | otherwise
    = error "bin: negatiivne argument"
```
, (140)

kus on arvesse võetud ka ülaltoodud kirjelduses käsitlemata jäänud, kuid kombinatoorselt mõttekas juht $n < k$ väärtusega 0. Aga, nagu öeldud, ei ole selline viis üksikuid binoomkordajaid arvutada kuigi mõistlik. Kui mõni definitsioon vajab järjest kõiki binoomkordajaid, tasub talle organiseerida lisaargument, mille algne väärtus on Pascali kolmnurk, rabagu sealt.

Lihtsamgi on aga kodeerida Pascali kolmnurka n-ö diagonaalide kaupa, kus üks diagonaal sisaldab kõik binoomkordajad C_n^k konstantse k ja suvalise $n \geqslant k$ jaoks, sest diagonaalid on lõpmatud ja pole tarvis tegelda lõputingimustega.

Jooksva diagonaali järgi järgmise leidmise kirjeldame koodis

```
osasummad
  :: (Num a)   ,
  => [a] -> [a]

osasummad (x : y : ys)
  = x : osasummad (x + y : ys)'
```
(141)

Pascali kolmnurga diagonaalide listi aga koodis

```
pasDiag
  :: [[Integer]]'

pasDiag
  = let
      bss
        = repeat 1 :                    .
          [osasummad bs | bs <- bss]
    in
    bss
```
(142)

Arvutuspõhimõte on nagu ennegi Pascali reegli iteratsioon.

Üksikute binoomkordajate leidmiseks muutuja `pasDiag` abil sobib definitsiooniga (140) samaväärne definitsioon

```
bin n k
  | n >= 0 && k >= 0
    = if n >= k then pasDiag !! (n - k) !! k else 0 .
  | otherwise
    = error "bin: negatiivne argument"
```

Vaatame lõpuks, kuidas Hanoi ketaste ülesande lahendust algse püstituse korral — tõstete jada leidmist — on võimalik optimeerida dünaamilise programmeerimisega.

Märkame, et arvutamisel varasemate definitsioonide (120) ja (121) järgi võivad lokaalse operaatori erinevad rekursiivsete pöördumiste ahelad viia samade argumentide komplektideni. Näiteks väljakutse argumentide väärtustega n, A, B, C toob kaasa väljakutsed argumentide väärtustega $n-1$, A, C, B ja argumentide väärtustega $n-1$, C, B, A, need aga omakorda tekitavad kumbki väljakutse argumentide väärtustega $n-2$, A, B, C.

Kuna rekursiivseid väljakutseid on eksponentsiaalne arv (igal tasemel tekib kaks uut), kuid võimalike argumentide kombinatsioonide arv on lineaarne, sest varraste ümberjärjestusi ehk permutatsioone on kokku ainult 6, siis on kopeerivate arvutuste osakaal väga suur.

Oleks vaja, et iga argumentide komplekti jaoks toimuks ainult üks väljakutse. Võimalikud argumentide komplektid rekursiivsetel pöördumistel võib paigutada $6 \times n$ tabelisse, kus n on algsisendi väärtus. Tabeli read vastavad varraste ümberjärjestustele. Kirjutame lokaalse operaatori selliselt, et tema töö tulemuseks argumendil väärtusega n on korraga terve selle tabeli veerg nr n (veerge nummerdame 0-st). Väljendame veerge 6-elemendiliste listidena. Elemendid selles listis vastaku järjest varraste ümberjärjestustele ABC, ACB, BAC, BCA, CAB, CBA. Varraste järjekord vastab argumentide järjekorrale varasemas koodis.

Selguse mõttes toome eraldi välja järgmise veeru leidmise operatsiooni koodiga

```
järgmHanoi
  :: [[String]] -> [[String]]'

järgmHanoi [hs1, hs2, hs3, hs4, hs5, hs6]
  = [
      hs2 ++ "AB" : hs6,
      hs1 ++ "AC" : hs4,
      hs4 ++ "BA" : hs5,                .
      hs3 ++ "BC" : hs2,
      hs6 ++ "CA" : hs3,
      hs5 ++ "CB" : hs1
    ]
```

Nüüd saame operaatori `hanoi` uuesti defineerida koodiga

```
hanoi n
  | n >= 0
    = let
        hanoi 0
          = replicate 6 []
        hanoi n                          ,      (143)
          = järgmHanoi (hanoi (n - 1))
      in
      head (hanoi n)
  | otherwise
    = error "hanoi: negatiivne argument"
```

mis töötab palju kiiremini isegi varasemast akumulaatoriga koodist (121). Kood on kiire vaatamata sellele, et kirjeldatud vahetulemuste struktuuris on vaid osa komponente vajalikud: näiteks järjekorras n-ndast 6-elemendilisest listist võetakse reaalselt ainult pea. Mittevajalikke komponente välja ei arvutata.

Aga kuna Hanoi tornide ülesandel on ainuüksi vastus algparameetri suhtes eksponentsiaalse suurusega, siis kõik käsitletud optimeeringud ja ka võimalikud muud jätavad töömahu paratamatult eksponentsiaalseks.

Ülesandeid

288. Jada t on defineeritud algtingimusega $t_0 = 1$ ja rekurrentse seosega $t_n = t_{n-1} + t_{\lfloor \frac{n}{2} \rfloor}$, mis kehtib iga $n > 0$ jaoks. (Tähis $\lfloor x \rfloor$ märgib arvu x täisosa.) Defineerida muutuja `t` väärtuseks funktsioon, mis mittenegatiivsetel täisarvulistel argumentidel annab tulemuseks vastava numbriga liikme jadas t. Arvutuse maht peab olema võrdeline argumendiga.

289. **Stirlingi esimest liiki** arvud s_n^k defineeritakse seostega

$$s_0^0 = 1, \quad \forall n > 0 \left(s_n^0 = 0\right), \quad \forall k > 0 \left(s_0^k = 0\right),$$
$$\forall n, k > 0 \left(s_n^k = s_{n-1}^{k-1} - (n-1) \cdot s_{n-1}^k\right).$$

Defineerida muutuja `s1` väärtuseks funktsioon, mis võtab sisse täisarvud n ja k ja annab välja arvu s_n^k, kui see on neil argumentidel defineeritud.

290. **Stirlingi teist liiki** arvud S_n^k defineeritakse seostega

$$S_0^0 = 1, \quad \forall n > 0 \left(S_n^0 = 0\right), \quad \forall k > 0 \left(S_0^k = 0\right),$$
$$\forall n, k > 0 \left(S_n^k = S_{n-1}^{k-1} + k \cdot S_{n-1}^k\right).$$

Defineerida muutuja `s2` väärtuseks funktsioon, mis võtab sisse täisarvud n ja k ja annab välja arvu S_n^k, kui see on neil argumentidel defineeritud.

291. Võttes konvolutsiooni kui jadade korrutamist, tähistame n-kordset konvolutsiooni nagu astendamist n-ga. Catalani jada ülaltoodud määratlus omandab siis seoste $c_0 = 1, \forall i > 0 \left(c_i = c_{i-1}^2\right)$ kuju. Osutub aga, et sobivad ka seosed

$$\begin{aligned} &\forall i \geqslant 0 \left(c_0^i = 1\right), \quad \forall j > 0 \left(c_j^0 = 0\right), \\ &\forall i > 0, j > 0 \left(c_j^i = c_j^{i-1} + c_{j-1}^{i+1}\right), \end{aligned} \tag{144}$$

kus astendajat võib lugeda kui teist indeksit ja kogu definitsiooni kui kahe muutuja rekurrentset võrrandit. (Esimese seose kehtivuse näitab lihtne induktsioon; teine seos on ilmne (polünoom astmes 0 on konstantselt 1). Viimane seos kehtib seetõttu, et astendamise definitsioonist ja jada c olemusest tulenevalt $c_j^i - c_{j-1}^{i+1} = (c^1 \cdot c^{i-1})_j - (c^2 \cdot c^{i-1})_{j-1} = c_0^1 \cdot c_j^{i-1} = c_j^{i-1}$.)

Kirjutada koodiga (137) antud muutujale `cats` uus samaväärne definitsioon seoste (144) baasil.

292. Sõne s **osasõneks** nimetatakse suvalist sõnet, mis on sõnest s saadud 0 või enama sümboli väljajätmisel (allesjäävad sümbolid omavahel kohti ei vaheta).

Defineerida muutuja `püos` väärtuseks funktsioon, mis võtab argumendiks kaks sõnet ja annab tulemuseks nende mingi pikima ühise osasõne.

5 Abstraheerimisvõimalused

5.1 Kõrgemat järku funktsioonid

Igasugune parameetrilisus tähendab abstraheerimist — mingi seose väljaeraldamist, mis kehtib ühtsena parameetri kõigi väärtuste jaoks. Iga objekti eraldi kirjeldamise asemel kirjeldatakse terve objektide pere ühtsel viisil. Seda eesmärki teenib programmeerimises enamik funktsioone.

Kõik eelnev on olnud vaid sissejuhatus, sest käsitlemata jäi väga oluline osa — kõrgemat järku funktsioonid. (Neist oli siiski siin-seal möödaminnes juttu.) Need funktsioonid esindavad abstraktsiooni kõrgemal tasemel. Vaadeldes funktsiooni esindavana teatavat käitumist, esindab kõrgemat järku funktsioon mingit üldisemat käitumist parameetrilisena teise käitumise suhtes. Pole raske mõista, et sellised funktsioonid võimaldavad ühekorraga kirjeldada mitmekesisemat arvutuskäikude valikut ja on laiema kasutusega kui funktsioonid, mille argumentide hulgas funktsioone pole.

Haskellis on muutujad, mille väärtuseks kõrgemat järku funktsioonid, enamasti ka polümorfsed, mis tähendab, et nad on kasutatavad paljude erinevate tüüpide jaoks. Polümorfism teeb nad veelgi universaalsemaks.

Kui konkreetne ülesanne või alamülesanne lahendub mõne standardse universaalse operaatori otsese rakendusena, tasub see võimalus ära kasutada, sest nii saadud kood on lugejale kergemini mõistetav.

Programmeerijal on võimalik lihtsasti ka endal muutujate väärtuseks meelepäraseid kõrgemat järku funktsioone defineerida.

5.1.1 Lihtsamad kõrgemat järku funktsioonid

1. **Varem käsitletud funktsioonid.** Oleme mõnda kõrgemat järku funkt-

siooni varem möödaminnes puudutanud: muutujate `map` ning `$` ja `$!` väärtuseks on nimelt kõrgemat järku funktsioonid. Need üksikud näited moodustavad kogu Haskelli võimalustest kõrgemat järku funktsioonide alal märksa vähem kui jäämäest tema veepealne osa. Ainuüksi mooduli Prelude kaudu on kasutatavad kümned ja kümned muutujad, mille väärtuseks kõrgemat järku funktsioonid.

Pangem ka tähele, et mõne avaldise väärtus võib sõltuvalt kontekstist olla esimest järku funktsioon või kõrgemat järku funktsioon. Sellised on kõik polümorfsed operaatorid, mis töötavad suvalise argumenditüübi korral. Ka selliseid muutujaid võib lugeja juba tunda.

Näiteks teame, et polümorfse muutuja `const` väärtuseks on funktsioon, mis võtab argumendiks mingi andme ja annab tulemuseks konstantselt seda annet välja andva funktsiooni. Argument võib olla suvalist tüüpi, muuhulgas ka funktsioonitüüpi. Seega `const` väärtus võib olla kõrgemat järku funktsioon, näiteks rakendus `const sin` on igati korrektne ja selle avaldise väärtus on funktsioon, mis suvalisel argumendil annab välja siinusfunktsiooni.

Lisades näiteks argumendi `5`, saame avaldise `const sin 5`, mille väärtus on järelikult siinusfunktsioon. Seega on see avaldis veel kord tüübikorrektselt rakendatav; näiteks avaldis `const sin 5 (pi / 2)` on samaväärne avaldisega `sin (pi / 2)`.

Näeme, et `const` ei pea üldjuhul sugugi ainult kahe argumendiga piirduma. Varieerides argumentfunktsiooni, võib talle sokutada lausa ükskõik kui palju argumente. Näiteks `const div 5` on samaväärne `div`-ga ja võtab seetõttu veel kaks argumenti, `const (\ _ _ _ -> 1) 5` võtab veel kolm argumenti jne.

Muuhulgas kõik põhilised andmestruktuurikonstruktorid on polümorfsed ja rakendatavad ka funktsioonitüüpi argumentidele. Tänu sellele on võimalik kirjeldada andmestruktuure, mille komponentideks funktsioonid.

Näiteks `(sin , cos)` on igati korrektne avaldis; siin on paarikonstruktori väärtus kõrgemat järku funktsioon.

Ülesandeid

293. Miks tekib viga, kui interaktiivses keskkonnas lasta väärtustada avaldis `(sin , cos)`? Kirjutada interaktiivses keskkonnas selline avaldis, milles `(sin , cos)` esineb alamavaldisena ja mille väärtustamine lõpeb normaalselt.

294. Kirjutada avaldis, milles tuntud operaatori `head` väärtus on karritatud funktsioon ja tema argumendi väärtus on lõpmatu list.

295. Kirjutada ülesande 294 lahendus nii, et lisaks lõpmatule listile on põhimõtteliselt võimalik lisada kuitahes palju üksikuid argumente.

2. Ümberpaigutavad funktsioonid. Kõige lihtsamate eeldefineeritud kõrgemat järku funktsioonide ülesandeks on funktsioonide argumente mingil viisil ümber paigutada või struktureerida või neid funktsioone mingis kindlas omavahelises suhtes kasutada.

Muutuja `flip` väärtus on funktsioon, mis võtab argumendiks karritatud funktsiooni f ja annab tulemuseks samuti karritatud funktsiooni, mis töötab nagu f, kuid võtab kaks esimest argumenti vastupidises järjekorras.

Näiteks avaldise `flip div` väärtus on täisarvulise jagamise funktsioon tavalisega võrreldes vahetatud argumentidega, st ta võtab jagaja enne jagatavat. Seega `flip div 3 17` väärtus on sama mis `div 17 3` väärtus ehk 5.

Kui on teada `flip` argumentfunktsiooni teine argument, siis saab `flip` asemel kasutada paremsektsiooni. (Süntaksianalüsaator tegelikult kirjutabki paremsektsioonid ümber `flip` kaudu, mis kajastub ka veateadetes.)

Näiteks avaldise `flip div 2` väärtus on funktsioon, mis jagab oma argumenti täisarvuliselt 2-ga, st sama mis sektsioonil `(‘div‘ 2)`.

Muutujate `curry` ja `uncurry` väärtuseks on teisendused funktsioonide karritatud ja järjendiargumendiga kuju vahel.

Näiteks avaldise `uncurry (+)` väärtus on funktsioon, mis võtab argumendiks arvupaari ja annab tulemuseks selle paari komponentide summa. Niisiis `uncurry (+) (2 , 3)` väärtus on 5, sest `(+) 2 3` väärtus on 5.

Avaldise `uncurry (flip div)` väärtus on aga funktsioon, mis võtab argumendiks täisarvupaari ja annab tulemuseks teise komponendi jagatise esimesega. Niisiis väärtustades `uncurry (flip div) (3 , 17)`, saame tulemuseks `5`.

Oleme eelnevas näinud, et muutujate `fst` ja `const` väärtuseks on sisuliselt sama operatsioon, lihtsalt teisel juhul on ta karritatud. Seega avaldised `curry fst` ja `const` on ekvivalentsed, samuti avaldised `uncurry const` ja `fst`.

Funktsiooni, mille rakendamine seisneb oma argumendile mingite funktsioonide järjestrakendamises, kusjuures need funktsioonid sellest argumen-

dist ei sõltu, nimetatakse nende funktsioonide **kompositsiooniks** (matemaatilise analüüsi terminoloogias **liitfunktsiooniks**).

Kompositsiooni leidmine kui kahe argumentfunktsiooniga operatsioon on defineeritud infiksoperaatori `.` väärtuseks. Kuna tegemist on karritatud funktsiooniga, siis rangelt võttes saab ta ühe funktsiooni f argumendiks ja annab tulemuseks funktsiooni, mis omakorda võtab argumendiks teise funktsiooni g ja annab välja f ja g kompositsiooni, st funktsiooni, mis oma argumendile rakendab kõigepealt funktsiooni g ja saadud tulemusele funktsiooni f. Muidugi on tüübikorrektsuseks tarvilik, et funktsiooni g väärtusetüüp võrduks funktsiooni f argumenditüübiga.

Näiteks avaldise `(+ 2) . (* 5)` väärtus on funktsioon, mis oma argumendi korrutab 5-ga ja liidab saadud vahetulemusele 2. Niisiis `((+ 2) . (* 5)) 7` väärtus on 37.

Eelnevas vaadeldud avaldis `uncurry (flip div)` on samaväärselt operaatoriga `.` ümber kirjutatav kujul `(uncurry . flip) div`.

Avaldis `(2 ^) . log` on aga tüübivigane, sest logaritmi väärtusetüüp on ujukomaarvutüüp, `(2 ^)` ootab aga argumendiks täisarvu.

Tihti läheb kompositsiooni vaja väljendamaks eitatud predikaati, st predikaati, mis igal argumendil annab välja originaalpredikaadi tulemuse eituse. Kui `p` väärtus on mingi predikaat, siis eitamiseks võib kirjutada `not . p`.

Näiteks sümbolitel töötavat predikaati, mis on tõene parajasti tühisümbolitest erinevatel sümbolitel, väljendab avaldis `not . isSpace`.

Ülesandeid

296. Küsida interaktiivses keskkonnas muutujate `flip`, `curry`, `uncurry`, `$`, `$!`, `.` tüübid ja saada neist aru.

297. Testida interaktiivses keskkonnas operaatorit `flip` argumentidega, mille väärtuseks on erinevad mittekommutatiivsed funktsioonid.

298. Veenduda interaktiivses keskkonnas testides, et `curry fst` töötab nagu `const` ja `uncurry const` töötab nagu `fst`.

299. Vaatleme lambdaavaldist `\ x y -> y`. Kirjutada veel kaks põhimõtteliselt erinevat sellesama väärtusega avaldist, mis sisaldavad ülimalt 10 sümbolit.

300. Kirjutada võimalikult lühike avaldis, mille väärtuseks on funktsioon, mille väärtus leitakse, võttes argumendist siinuse ja saadud tulemusest koosinuse. Testida interaktiivses keskkonnas.

301. Leida täisargumenteeritud tüübikorrektne avaldis, milles muutujad `uncurry` ja `flip` oleksid järjest rakendatud nii, et `uncurry` rakenduks esimesena. Kirjutada seejärel avaldis samaväärselt ümber, lisades ka kompositsiooni.

5.1.2 Rekursiooniskeeme abstraheerivad funktsioonid

Paljud kõrgemat järku funktsioonid abstraheerivad teatavaid tüüpilisi rekursiooniskeeme. See võimaldab paljude omavahel sarnase ülesehitusega rekursiivsete definitsioonide asemel kirjutada definitsioonid ilma rekursioonita mingi kõrgemat järku funktsiooni rakendamise kaudu.

Näiteks võiks programmeerija iga ülesande, kus on vaja mingit funktsiooni listi igale elemendile rakendada, lahendada omaette rekursiivse definitsiooniga, kuid operaatori `map` kasutamine päästab ta sellest vaevast.

1. Itereerivad funktsioonid. Siin vaatame funktsioone, mille ülesandeks on oma argumentfunktsiooni korduv järjestrakendamine.

Muutuja `iterate` väärtus on kõrgemat järku funktsioon, mis võtab argumendiks sellise funktsiooni f, mille argumendi- ja väärtusetüüp on üks ja sama, ja annab tulemuseks funktsiooni, mis võtab sama tüüpi andme x argumendiks ja annab välja lõpmatu listi elementidega x, $f\,x$, $f(f\,x)$ jne.

Näiteks avaldise `iterate (* 2) 1` väärtuseks on lõpmatu list kõigi 2 astmetega, st $1:2:4:8:16:\ldots$.

Ilma operaatorita `iterate` peaks selliseid liste kirjeldama listirekursiooniga või akumulaatorite abil.

Muutujat `iterate` kasutades võiksime näiteks deklaratsiooniga (87) või (89) antud muutuja `geom` samaväärselt defineerida koodiga

```
geom a q
  = iterate (* q) a
```

Ka Fibonacci jada on analoogiliselt kirjeldatav, ainult et kuna iga liige määratakse kahe, mitte ühe eelmise poolt, siis tuleb itereerida paaride peal ja pärast võtta neist esimesed komponendid. Seetõttu pole tegemist just kõige efektiivsema viisiga Fibonacci jada arvutamiseks, ehkki töömaht on lineaarne. Koodiks tuleb

```
fibs
  = map fst
    (iterate (\ (a , b) -> (b , a + b)) (0 , 1))
```
. (145)

(Liiga pika rea vältimiseks on avaldise `map fst` argument viidud järgmisele reale; see on Haskelli koodipaigutusreeglite järgi lubatud.) Paarid listis, mille iteratsiooniprotsess tekitab, on parajasti kõik kahest järjestikusest Fibonacci arvust koosnevad paarid. Avaldise `map fst` rakendamine kaotab teised komponendid ja järele jääbki Fibonacci arvude jada.

Koodiga (139) andsime muutuja `pasRead` väärtuseks lõpmatu listi, mille elementideks Pascali kolmnurga read. Ka see kood määratles itereeriva arvutuse, kusjuures vahetulemused on listid. Operaatori `iterate` abil saab anda lühema samaväärse definitsiooni

```
pasRead
  = iterate kahesummad' [1]'
```
(146)

kus `kahesummad'`, mida ka eelmine variant kasutas, on antud koodiga (138).

Mõneti sarnane samuti iteratiivset käitumist abstraheeriv kõrgemat järku funktsioon on muutuja `until` väärtuseks. Tema võtab järjest argumentideks predikaadi p, selle predikaadi argumenttüübist samasse tüüpi töötava funktsiooni f ning sama tüüpi argumendi x ning annab välja esimese liikme jadas $x, f\,x, f(f\,x)\ldots$, mis rahuldab predikaati p, kui seal selline leidub, vastasel korral on väärtuseks $\perp$ (arvutus jääb lõpmatusse tsüklisse).

Näiteks avaldise `until (> 100) (* 2) 1` väärtus on 128, sest 128 on esimene arvu 2 aste, mis on suurem 100-st.

Ülesandeid

302. Küsida interaktiivses keskkonnas muutujate `iterate` ja `until` tüübid ja saada neist aru.

303. Kasutades erisüntaksi asemel muutujat `iterate`, kirjutada avaldis, mille väärtuseks on mingi aritmeetiline jada.

304. Kirjutada avaldis, mille väärtuseks on lõpmatu list, mille elemendid on järjest kõik ainult suurtest A-tähtedest koosnevad lõplikud sõned alustades tühjaga.

305. Anda kõrgemat järku funktsioonide abil uued lahendused ülesannetele 220, 221 ja 223.

306. Arvutada interaktiivses keskkonnas kirjutatud ühe avaldise abil esimene arvu 18 aste, mis jagub 486-ga.

307. Muutuja `until` abil defineerida muutuja `collatzPikkus` väärtuseks funktsioon, mis positiivsel täisarvulisel argumendil n annab tulemuseks elementide arvu n-ga algavas Collatzi jadas.

2. Kontrollivad ja selekteerivad funktsioonid. Omaette rühma moodustavad kõrgemat järku funktsioonid, mis võtavad argumendiks predikaadi ja listi, kontrollivad predikaadi kehtivust listi elementidel ja koostavad selle analüüsi põhjal ühe- või teistsuguse tulemuse.

2.1. Ainult kontrollivad funktsioonid. Muutujate `all` ja `any` väärtuseks on funktsioonid, mis võtavad argumendiks mingil tüübil töötava predikaadi p ja annavad tulemuseks sellist tüüpi elementidega listidel töötava predikaadi. Täpsemalt, `all` korral tuleb True siis, kui list on lõplik ja iga element rahuldab predikaati p, ja False siis, kui mõni element predikaati p ei rahulda; `any` puhul tuleb True siis, kui listi mõni element rahuldab predikaati p, ja False siis, kui list on lõplik ja ükski element predikaati p ei rahulda.

Kui listi mingi element otsustab üksi funktsiooni väärtuse (`all` puhul predikaati mitte rahuldav ja `any` puhul rahuldav element), siis arvutamisel sellest kaugemale listi ei loeta.

Näiteks avaldise

```
all (not . isSpace)                                    (147)
```

väärtus on funktsioon, mis võtab argumendiks sõne ja annab tulemuseks True parajasti siis, kui selle sõne ükski sümbol pole tühisümbol (ja ta on lõplik).

Ülesandeid

308. Kirjutada avaldis (147) samaväärselt ümber operaatori `any` abil.

2.2. Omaduse põhjal selekteerivad funktsioonid. Muutuja `takeWhile` väärtus on funktsioon, mis võtab järjest argumendiks predikaadi ja listi, mille elemenditüüp võrdub predikaadi argumenditüübiga, ning annab tulemuseks listi pikima algusosa, mille kõik elemendid rahuldavad predikaati.

Näiteks avaldise `takeWhile (< 100) (iterate (* 2) 1)` väärtus on lõplik list elementidega 1, 2, 4, 8, 16, 32, 64, sest need elemendid 2 astmete listi alguses on 100-st väiksemad, järgmine element 128 aga pole.

Avaldise `takeWhile (> 100) (iterate (* 2) 1)` väärtus on aga tühi list, sest juba esimene element 1 pole suurem 100-st, st predikaati ei rahulda. See ei loe, et tagapool on ka predikaati rahuldavaid elemente, sest otsitakse listi algusosa.

Muutuja `dropWhile` väärtuseks on funktsioon, mis võtab samad argumendid nagu eelmine, kuid annab neil tulemuseks argumentlisti selle osa, mis `takeWhile` puhul üle jääb.

Seega avaldise `dropWhile (< 100) (iterate (* 2) 1)` väärtus on lõpmatu list $128 : 256 : 512 : \ldots$, `dropWhile (> 100) (iterate (* 2) 1)` väärtus aga kogu 2 astmete list, võrdne `iterate (* 2) 1` väärtusega.

Ka muutuja `filter` väärtuseks on funktsioon, mis võtab samad argumendid nagu kaks eelnevalt vaadeldud funktsiooni, kuid tema annab välja listi kõigist argumentlisti elementidest, mis predikaati rahuldavad.

Avaldise `filter (> 100) (iterate (* 2) 1)` väärtus on lõpmatu list $128 : 256 : 512 : \ldots$, sest predikaadi väärus mingil elemendil ei lõpeta arvutust ära.

Avaldise `filter isUpper "Tere, Haskell!"` väärtus on "TH"; võrdluseks `takeWhile isUpper "Tere, Haskell!"` annab väärtuseks "T" ja `dropWhile isUpper "Tere, Haskell!"` annab "ere, Haskell!".

Avaldis `filter isUpper [1 .. 100]` on aga tüübivigane, sest predikaadi argumenditüüp on sümbolitüüp, kuid listi elementitüüp arvuline.

Muutujate `takeWhile` ja `dropWhile` rakendamisel loetakse listi ainult niikaua, kuni leitakse predikaati mitte rahuldav element. Muutuja `filter` korral vaadatakse aga listi kõik elemendid läbi; kui list on lõpmatu ja teatud kohast alates on predikaat igal elemendil väär, tekib tühi lõpmatu tsükkel.

Näiteks avaldise `filter (< 100) (iterate (* 2) 1)` väärtus on osaline list $1 : 2 : 4 : 8 : 16 : 32 : 64 : \bot$, pärast elemendi 64 leidmist jääb arvutus tsüklisse.

Pisut keerulisema näitena kirjutame nüüd koodi, mis defineerib muutuja `algs` väärtuseks kõigi algarvude listi. Osutub, et sobib definitsioonike

```
algs
  = let
      isAlg n
        = all ((/= 0) . (n `mod`))        ,
          (takeWhile ((<= n) . sqr) algs)
    in
    2 : filter isAlg [3 .. ]
```

signatuuriks võib panna

```
algs
  :: [Integer]
```

Muutuja `sqr` väärtuseks peab olema eraldi (näiteks koodiga (37)) defineeritud ruutfunktsioon.

Idee on järgmine. Teame, et 2 on algarv. Edasi kasutame tuntud fakti, et iga kordarv n jagub mingi algarvuga, mis pole suurem kui $\sqrt{n}$. Seega võib sõelale jätta parajasti need 2-st suuremad täisarvud n, mis ei jagu ühegi sellise algarvuga, mille ruut pole suurem kui n. Operaatori `filter` argumendiks oleva muutuja `isAlg` väärtuseks on lokaalselt defineeritud just seda kontrolliv predikaat. Tõepoolest: argumendil n ta kontrollib, kas algarvude listi algusjupis, milles olevate arvude ruut on ülimalt n, rahuldab iga element tingimust, et n jagamisel temaga saadav jääk on nullist erinev.

Tegemist on listirekursiooniga: muutuja `algs` definitsioon pöördub `algs` enda poole. Kood töötab, kuna iga järgmise arvu kontrollimisel läheb vaja ainult neid algarve, mis on selleks ajaks juba leitud.

Ülesandeid

309. Defineerida muutuja `ilmaVäikesteta Algus` väärtuseks funktsioon, mis argumenttekstil annab välja tema pikima väiketähti mitte sisaldava algusjupi.

310. Arvutada interaktiivse keskkonna käsureale kirjutatud ühe avaldise abil, mitu arvu 3 astet on ülimalt 100-kohalised.

311. Kasutades ülesande 247 lahendust, kirjutada avaldis, mille väärtuseks on list kõigist paaritutest kolmnurkarvudest.

312. Defineerida muutuja `elimTaanded` väärtuseks funktsioon, mis võtab argumendiks sõnede listi ja annab tulemuseks nende sõnede algusest tühisümbolite jorude kaotamisel järele jäävate sõnede listi.

313. Defineerida muutuja `elimJärjKorduvad` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks listi, mis erineb argumentlistist parajasti selle poolest, et järjestikused võrdsed elemendid on asendatud ühe esinemisega.

314. Defineerida muutuja `elimKorduvad` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks listi, milles esinevad parajasti kõik argumentlisti elemendid, kuid igaüks vaid ühe korra.

315. Kirjutada tüübikorrektne avaldis, kus muutuja `filter` argumendiks oleks `snd` ja järgmiseks argumendiks mõni mittetühi list.

2.3. Omaduse järgi jaotavad funktsioonid. Operaator `span` ühendab endas `takeWhile` ja `dropWhile` tegevuse, andes mõlema tulemused välja paarina.

Näiteks avaldise `span (< 100) (iterate (* 2) 1)` väärtus on lõplik list elementidega 1, 2, 4, 8, 16, 32, 64 paaris listiga 128 : 256 : 512 : . . ..

Moodulist Data . List tuleb veel analoogiline operaator `partition`, mille aluseks on `takeWhile` asemel `filter`. Tema väärtuseks on funktsioon, mis võtab samasugused argumendid ning annab tulemuseks listipaari, kus argumentlisti elemendid on vastavalt argumentpredikaadi kehtivusele kaheks jaotatud: elemendid, millel predikaat kehtib, on esimeses ning ülejäänud elemendid teises listis.

Näiteks on avaldise `partition isUpper "Tere, Haskell!"` väärtuseks paar (“TH”, “ere, askell!”).

Ülesandeid

316. Anda koodiga (73) või (74) defineeritud muutujale `kaheksLahuta` samaväärne lihtne definitsioon kõrgemat järku funktsiooni abil.

3. Kujutavad funktsioonid. Lugeja on juba tuttav operaatoriga `map`, mille abil rakendati listi igale elemendile mingit üht ja sama funktsiooni. Lihtsamad listikomprehensiooniga avaldised on võimalik niisama lihtsalt ümber kirjutada `map` abil.

Näiteks avaldised (46) ja (47) on vastavalt samaväärsed avaldistega

```
map (\ x -> x * x) [0 .. ],
```

```
map (\ x -> (x , x * x)) [1 .. 9].
```

Operaatori `map` kasutamisel on see eelis, et funktsioon, mis listile rakendatakse, on ilmutatud kujul alamavaldisena esitatud.

Toome veel näite operaatori `map` kasutamisest rekursiooniskeemis, ehkki ka siin pole listikomprehensiooni tundmise puhul midagi põhimõtteliselt uut.

Listi l **osalistiks** nimetatakse listi, mille elementideks on mingi suvaline valik l elementidest ilma nende järjekorda muutmata. Olgu meil vaja defineerida muutuja `osalistid` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks tema kõigi osalistide listi. Eesmärgiks olgu läbi käia just kõik võimalikud positsioonide valikud ja koostada listid vastavatel positsioonidel olevaist andmetest, nii et kui argumentlistis esineb kordusi, siis tekivad kordused ka tulemusse. Siis sobib signatuur

```
osalistid
  :: [a] -> [[a]]'
```

mis elemendituubile kitsendusi ei sea, kuna kordumisi kontrollida pole vaja.

Lahenduse aluseks on tähelepanekud, et mittetühja listi l osalistid on parajasti l saba osalistid ja lisaks needsamad saba osalistid koos listi l peaga, tühjal listil on aga parajasti üks osalist — tema ise. Mittetühja argumentlisti puhul tuleb niisiis vastus koostada rekursiivse pöördumise tulemusest listi sabal, võttes tema elemendid üks kord niisama, teisel korral aga lisada kõigi nende ette argumentlisti pea. Viimatimainitud muudatuse tegemiseks kasutamegi operaatorit `map`. Et mitte arvutada listi saba osalistide listi mitu korda välja, toome tema salvestamiseks sisse lokaalse muutuja `pool`. Saame definitsiooni

```
osalistid (x : xs)
  = let
      pool
        = osalistid xs
    in
    map (x :) pool ++ pool
osalistid _
  = [[]]
```
. (148)

Operaatori `map` analoog, mis tegutseb korraga kahel listil, on `zipWith`. Tema väärtus on funktsioon, mis võtab järjest argumentideks mingi karritatud funktsiooni ja kaks listi, tulemuseks aga annab ühe listi, mille elemendid on saadud argumentlistide vastavate elementide kokkuarvutamisel selle funktsiooniga. Kui üks list on teisest pikem, siis tema ülejäävat osa ignoreeritakse.

Näiteks `zipWith (*) [2, 3] [5, 7, 11]` väärtus on 2-elemendiline list elementidega 10, 21. Tulemuslisti esimene element 10 saadakse kui argumentlistide esimeste elementide 2 ja 5 korrutis, tulemuse teine element 21 aga kui nende teiste elementide 3 ja 7 korrutis ning et esimeses argumentlistis rohkem midagi pole, siis lõpeb siin ka tulemuslist.

Seega operaatori `zipWith` töö on analoogiline operaatoriga `zip`, milles paaridesse panek on asendatud suvalise binaarse kokkuarvutamisega. Teisi sõnu, `zip` l_1 l_2 ja `zipWith (,)` l_1 l_2 on alati samaväärsed.

Samas saab ka `zipWith` täisargumenteeritud rakenduse samaväärselt ümber kirjutada `zip`, `map` ja `uncurry` abil: iga $\oplus$ ning l_1, l_2 korral on avaldised `zipWith` $\oplus$ l_1 l_2 ja `map (uncurry (` $\oplus$ `))` `(zip` l_1 l_2`)` samaväärsed.

Operaatori `zipWith` abil saab elegantselt realiseerida arvutusi, mis nõuavad listi kahe järjestikuse elemendi vaatlemist igal rekursioonisammul.

Näiteks operaatori `kahesummad` definitsioon (75) on mõneti kohmakas. Kuid `zipWith` abil on `kahesummad` lihtsasti defineeritav koodiga

```
kahesummad xs
  = zipWith (+) xs (tail xs)
```
(149)

Et `tail xs` tulemuseks on argumentlisti nihe ühe koha võrra, siis listide vastavad elemendid, mis kokku liidetakse, ongi parajasti algses listis järjestikused.

Pangem tähele, et definitsioon (149) töötab ka juhul, kui `xs` väärtus on tühi list ja `tail xs` seega vigane. Põhjus peitub selles, et `zipWith` arvutamisel uuritakse argumentliste ainult niikaua, kui üks neist ära lõpeb, kusjuures esimest listi kontrollitakse selles suhtes enne.

Operaatorit `kahesummad` läks vaja Fibonacci jada arvutamiseks listirekursiooni abil definitsiooniga (92). Asendades seal funktsiooni `kahesummad` rakendamise definitsiooni (149) parema poole järgi, saame Fibonacci jadale uue definitsiooni

```
fibs
  = 0 : 1 : zipWith (+) fibs (tail fibs)'
```

mille põhimõte on sama mis koodil (92). Pisut kohendades saame elegantsema definitsiooni

```
fibs@ (_ : fs)
  = 0 : 1 : zipWith (+) fibs fs
```
(150)

Ülesandeid

317. Küsida interaktiivses keskkonnas muutuja `zipWith` tüüp ja saada sellest aru.

318. Kirjutada avaldis, mille väärtus on funktsioon, mis võtab argumendiks sõne ja annab tulemuseks tema iga tähe suureks teisendamisel saadava sõne.

319. Ilma komprehensioonsüntaksita arvutada sõne koodide järjekorras kõigist sümbolitest, mis kooditabelis leiduvad.

320. Ilma komprehensioonsüntaksita defineerida muutuja `vahedeKorrutis` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab väärtuseks tema mistahes kahest erinevast kohast võetud elementide vahede (eespoolsest lahutatakse tagapoolne) korrutise.

321. Kirjutada avaldis, mille väärtuseks on lõpmatu list lõpmatutest listidest. Anda see avaldis interaktiivses keskkonnas argumendiks sellisele avaldisele, et tulemusena leitakse igast listist mõned elemendid.

322. Defineerida muutuja `nurk` väärtuseks karritatud funktsioon, mis võtab argumendiks täisarvu n ja listide listi l, mida interpreteerime kahemõõtmelise tabelina, ja annab tulemuseks selle tabeli vasaku ülemise nurga mõõtmetega $n \times n$ samamoodi listide listi kujul. Näiteks kui `nurk` argumentideks panna `10` ja ülesande 161 lahenduseks olev avaldis, peab tulemuseks olema 10×10 korrutustabel.

323. Kirjutada definitsioon (149) samaväärselt ümber, võttes eeskuju definitsioonist (150): `tail` poole pöördumise asemel tuleb argumentlisti saba saada näidisesobitusest.

324. Kasutades operaatorit `zipWith`, defineerida muutuja `indeksid` väärtuseks funktsioon, mis võtab argumendiks suvalise listi ja annab tulemuseks listi, mis on niisama pikk kui argumentlist ja mille elemendid on järjestikused naturaalarvud alates 0-st.

325. Defineerida muutuja `selekt` väärtuseks karritatud funktsioon, mis võtab argumendiks tõeväärtuste listi ja arvude listi ja annab tulemuseks listi, milles neil positsioonidel, kus tõeväärtuste listis on `True`, on algse arvude listi vastavad elemendid, mujal aga nullid. Võib eeldada, et argumentlistid on ühepikkused (vastasel korral võib pikema listi ülejäävat osa ignoreerida).

326. Lahendada kõrgemat järku funktsioonide abil ülesanne 196.

327. Lahendada kõrgemat järku funktsioonide abil ülesanded 246 ja 247.

328. Defineerida muutuja `diagPööre` väärtuseks funktsioon, mis võtab argumendiks listide listi l: kui l on lõpmatu list lõpmatutest listidest, siis annab väärtuseks lõpmatu listi lõplikest listidest, milles esimene sisaldab parajasti esimese listi esimest elementi, teine sisaldab esimese listi teise ja teise listi esimese elemendi, kolmas esimese listi kolmanda, teise teise ja kolmanda esimese elemendi jne.

    Kasutades seda operaatorit ning koodiga (141) ja (142) defineeritud muutujat `pasDiag`, anda koodiga 139 või 146 defineeritud muutujale `pasRead` uus samaväärne definitsioon, mis ei kasuta abimuutujat `kahesummad'`.

329. Kui `zipWith` argumendi väärtus on kommutatiivne funktsioon, kas siis teeb alati sama välja, ükskõik kummas järjekorras anda listid?

4. Kokkuarvutavad funktsioonid.

4.1. Vahetulemusi mitte salvestavad funktsioonid. Muutuja `foldl` väärtus on funktsioon, mis võtab argumentideks järjest binaarse operatsiooni $\oplus$, andme e ja listi l ning mille väärtuse võib saada, kui alustada andmest e

ja igal sammul arvutada jooksev vahetulemus paremalt operatsiooni $\oplus$ abil kokku listi l järjekordse elemendiga, läbides nii listi kõik elemendid algusest lõpuni. See tähendab, et kui l elemendid on $a_1, \ldots, a_n$, on tulemus

$$(\ldots((e \oplus a_1) \oplus a_2) \oplus \ldots) \oplus a_n. \tag{151}$$

Erijuhul, kui list on tühi, on tulemuseks lihtsalt e.

Näiteks avaldise `foldl (+) 0 [1, 2, 3]` väärtus on 6, sest $0+1+2+3=6$. Avaldise `foldl (-) 0 [1, 2, 3]` väärtus on -6, sest nüüd tuleb arvutada $0-1-2-3$. Avaldise `foldl (++) [] [[1, 3], [7], []]` väärtus on analoogselt lõplik list elementidega 1, 3, 7.

Muutuja `foldr` väärtus on funktsioon, mis võtab argumentideks järjest operatsiooni $\oplus$, andme e ja listi l ning mille väärtuse võib saada, kui alustada andmest e ja igal sammul arvutada jooksev vahetulemus vasakult operatsiooni $\oplus$ abil kokku listi l järjekordse elemendiga, läbides nii listi kõik elemendid lõpust alguseni. Kui l elemendid on $a_1, \ldots, a_n$, on tulemus

$$a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus e)\ldots)). \tag{152}$$

Erijuhul, kui list on tühi, on siingi tulemuseks lihtsalt e.

Näiteks avaldise `foldr (+) 0 [1, 2, 3]` väärtus on 6, sest $1+2+3+0=6$, st tulemus on sama mis samade argumentidega `foldl` puhul.

Samas avaldise `foldr (-) 0 [1, 2, 3]` väärtus on mitte -6 nagu `foldl` puhul, vaid hoopis 2, sest $1-(2-(3-0)) = 1-(2-3) = 1-(-1) = 2$.

On ilmne, et assotsiatiivse kommutatiivse operatsiooni ja lõpliku listi puhul annavad `foldl` ja `foldr` alati sama tulemuse.

Et esitustes (151) ja (152) on listi elemendid omavahel samas järjestuses, siis on tulemused võrdsed ka juhul, kui operatsioon on assotsiatiivne ja mitte tingimata kommutatiivne, kuid e on selle operatsiooni ühikelement (st lisaoperatsioon temaga ükskõik kummalt poolt tulemust ei muuda).

Näiteks ka avaldise `foldr (++) [] [[1, 3], [7], []]` väärtuseks on lõplik list elementidega 1, 3, 7.

Kuid isegi võrdse tulemuse puhul ei ole alati ükskõik, kumba operaatorit kasutada, sest arvutus võib ühel juhul olla aeglasem kui teisel.

Viimases näites `foldr` korral tehakse kõigepealt tühja listi konkateneerimisel 0 ümbertõstmist, siis 1-elemendilise listi konkateneerimisel 1 ümbertõstmine ja 2-elemendilise listi konkateneerimisel 2 ümbertõstmist; kokku tuleb 3 ümbertõstmist.

Operaatori `foldl` puhul ehitatakse listi vasakult paremale: alguses konkateneeritakse tühi list (`foldl` teise argumendi väärtus) 2-elemendilise ette, seejärel saadud 2-elemendiline 1-elemendilise ette ja lõpuks konkateneeritakse saadud 3-elemendiline list tühja listi ette. Siin tuleb kokku $0+2+3=5$ ümbertõstmist. Esimese listi elemente tõstetakse ümber kaks korda.

Kerge on mõista, et mida rohkem liste listis on, seda rohkem kordi tõstetakse alguses paiknevate listide elemente ümber. Operaatori `foldr` korral tühiümbertõstmisi pole, kõik elemendid satuvad kohe oma õigesse kohta. Seega konkateneerimise korral tuleb eelistada operaatorit `foldr`.

Vaatamata esituste (151) ja (152) struktuuri selgele erinevusele toimub listi elementide arvutusse kaasahaaramine nii `foldl` kui ka `foldr` puhul samas järjekorras — listi algusest lõpu suunas.

Operaatori `foldl` puhul on see selge. Kuid ka `foldr` puhul, kui argumentide väärtused on näiteks $\oplus$, e ja l, siis tehakse kindlaks, kas l on mittetühi, ja kui nii, siis püütakse kohe arvutada $a_1 \oplus x_1$, kus a_1 on tema esimene element ja x_1 kõigi ülejäänud elementide ja e kokkuarvutuse tulemus, mis üldiselt pole veel teada; x_1 arvutamiseks omakorda vaadatakse, kas listis elemente veel on, ja kui nii, siis püütakse arvutada $a_2 \oplus x_2$, kus a_2 on teine element ja x_2 kõigi järgnevate elementide ja e kokkuarvutuse tulemus, jne.

Kui juhtub, et $a_i \oplus x_i$ leidmiseks pole annet x_i üldse vaja, siis listi edasi ei uurita. Samuti võib juhtuda, et $a_i \oplus x_i$ on andmestruktuur, mille algusosa saab leida ilma x_i-d teadmata — siis see algusosa pannakse paika enne x_i arvutamist. Nii võib operaator `foldr` anda ka lõpmatul listil lõpliku ajaga tulemuse. Lõpmatu listi puhul `foldr` teisest argumendist midagi ei sõltu, selleni arvutus kunagi ei jõua. Lõpptulemust võib (152) asemel väljendada

$$a_1 \oplus (a_2 \oplus (a_3 \oplus (\ldots\ldots))). \tag{153}$$

Kuna `const` rakendamisel kahele argumendile arvutus teist argumenti ei kasuta, siis näiteks `foldr const 0 [1 .. ]` väärtus on 1 (listi esimene element).

Aritmeetilised operatsioonid on realiseeritud agaratena, mistõttu avaldised kujul `foldr (*) 1 l`, mille väärtus võiks olla `l` väärtuseks oleva listi kõigi elementide korrutis, tegelikult normaalset väärtust ei oma, juhul kui `l` väärtus on lõpmatu list. Samas on loomulik soov, et kui list sisaldab nulle, antaks tulemuseks 0. See saab täidetud, kui paneme `foldr` argumendiks `(*)` asemel `\ x p -> if x == 0 then 0 else x * p`.

Seevastu kui operaatori `foldl` viimase argumendi väärtus on lõpmatu list, jääb arvutus alati lõpmatusse tsüklisse ilma midagi välja andmata. Põhjus on selles, et sulgude paigutamisel vasakult listi elementide ümber on kõik alamavaldised lõplikud ja isegi kui laisa väärtustamise tõttu saab neist midagi välja jätta, jääb ikka järele veel lõpmata palju sooritamata tehteid.

Operaatorite `foldl` ja `foldr` abil saab palju rekursiivseid definitsioone tunduvalt lühendada.

Näiteks koodiga (105) defineeritud muutuja `suurSumma` (väärtuseks funktsioon, mis argumendil n leiab arvude 1 kuni n neljandate astmete summa) võiks samaväärselt defineerida ka koodiga

```
suurSumma n
  | n >= 0
    = foldl (\ a i -> a + i ^ 4) 0 [1 .. n]
  | otherwise
    = error "suurSumma: neg. liidetavate arv"
```
. (154)

Operaatori `foldl` kasutamisega kaasneb siiski puudus, et vahetulemusi ei väärtustata jooksvalt. Jooksvaks väärtustamiseks tuleb `foldl` asemel kasutada moodulist Data . List saadavat operaatorit `foldl'`.

Operatsioonil, millega listi elementide kokkuarvutus toimub, ei pea üldjuhul olema mõlemad argumendid sama tüüpi. Nõutav on vaid see, et operatsiooni väärtusetüüp võrduks vastavalt kas tema esimese argumendi tüübiga (`foldl` puhul) või teise argumendi tüübiga (`foldr` puhul). Sama tüüpi peab ühtlasi olema ka ülal e-ga tähistatud algväärtus.

Näitena anname koodiga (148) defineeritud muutujale `osalistid` uue, konkatenatsioonivaba definitsiooni

```
osalistid (x : xs)
  = let
      pool
        = osalistid xs
    in
    foldr (\ as ass -> (x : as) : ass) pool pool
osalistid _
  = [[]]
```
. (155)

Erinevus on ainult avaldises võtmesõna **`in`** järel, kus listi saba osalistide listi järgi pannakse kogu listi osalistide list kokku varasemast erinevalt. Lisatud peaga listide

listi omaette ei moodustata, vaid need listid pannakse kohe oma lõplikku asupaika saba osalistide listi ees.

Koodis (155) pole `foldr` argumentfunktsiooni argumentide tüübid samad: esimese argumendi tüüp on võrdne teise argumendi elementitüübiga. Rakendamine on tüübikorrektne, sest avaldised `ass`, `(x : as) : ass` ja `pool` on sama tüüpi.

Kuigi `foldr` aitas kõrvaldada konkatenatsioonidest tulenevad tühiümbertõstmised, on sellest tulenev võit kiiruses siin tagasihoidlik. Arvutuse maht on nii definitsiooni (148) kui ka (155) puhul eksponentsiaalne, kuna juba tulemuslisti pikkus on algse listi pikkuse suhtes eksponentsiaalne.

Ülesandeid

330. Küsida interaktiivses keskkonnas muutujate `foldl`, `foldr` tüübid ja saada neist aru.

331. Anda koodiga (154) defineeritud muutujale `suurSumma` samaväärne definitsioon `foldr` kaudu.

332. Anda muutujale `suurSumma` koodiga (154) sarnane samaväärne definitsioon, mille järgi arvutades väärtustatakse vahetulemused jooksvalt.

333. Anda koodiga (133) defineeritud muutujale `hanoiSeisud` uus definitsioon, mis väldib tühiümbertõstmisi.

334. Leida sellised argumendid $\mathfrak{h}$ ja $\mathfrak{e}$, et avaldise `foldr` $\mathfrak{h}$ $\mathfrak{e}$ $\mathfrak{l}$ väärtustamine lõpetab mingi $\mathfrak{l}$ korral, mille väärtus on lõpmatu list, töö normaalselt lõpliku ajaga, kuid avaldise `foldr (flip` $\mathfrak{h}$`)` $\mathfrak{e}$ $\mathfrak{l}$ väärtustamine jääb iga $\mathfrak{l}$ korral, mille väärtus on lõpmatu list, tsüklisse ilma midagi välja andmata.

335. Võidakse arvata, et vahetulemusi saab panna jooksvalt väärtustama, kui kirjutada `suurSumma` definitsioon (154) ümber kujule

```
suurSumma n
  | n >= 0
    = foldl (($!) (\ a i -> a + i ^ 4)) 0 [1 .. n] .
  | otherwise
    = error "suurSumma: neg. liidetavate arv"
```

Miks see arvamus ei ole õige?

4.2. Vahetulemusi salvestavad funktsioonid. Operaator `scanl` töötab muus osas operaatori `foldl` moodi, kuid salvestab kõik vahetulemused järjest listi; see list on ka väljaantavaks väärtuseks. Operaator `scanl` töötab normaalselt ka lõpmatu listi peal, sest ehitab pidevalt tulemuslisti uusi lülisid.

Näiteks definitsiooniga (104) antud muutuja `suuredSummad` (väärtuseks lõpmatu list, kus positsioonil n on arvude 1 kuni n neljandate astmete summa) saab samaväärselt defineerida koodiga

```
suuredSummad
  = scanl (\ a i -> a + i ^ 4) 0 [1 .. ].
```

Operaator `scanl` võimaldab defineerida Fibonacci arvude listi koodiga

```
fibs
  = 0 : scanl (+) 1 fibs'
```
(156)

mis on elegantsem kõigist senivaadelduist.

Anname ka Pascali kolmnurga diagonaalide listile abimuutujateta ühereal ise definitsiooni. Selleks kirjutame kõigepealt varasema definitsiooni (142) parema poole samaväärselt ümber avaldisega `iterate osasummad (repeat 1)`, kus abimuutuja `osasummad` on defineeritud koodiga (141). Kuid `osasummad` väärtus on funktsioon, mis leiab elementide summad listi kõigis algusjuppides, seda aga saab arvutada `scanl` abil.

Pisut tegemist on seetõttu, et `scanl` puhul liidetakse listi elementidele juurde ka teisest argumendist saadav algväärtus, kusjuures esimene summa ainult sellest koosnebki. Võttes algväärtuseks 0, see summasid ei mõjuta, kuid ülearusest nullsummast vabanemiseks peab tulemusele rakendama operaatorit `tail`.

Seega `osasummad` on samaväärne avaldisega `tail . scanl (+) 0`. Kokku saame soovitud definitsiooniks

```
pasDiag
  = iterate (tail . scanl (+) 0) (repeat 1).
```
(157)

On olemas ka operaator `scanr`, mis vastab operaatorile `foldr` samamoodi nagu `scanl` operaatorile `foldl`.

Ülesandeid

336. Küsida interaktiivses keskkonnas muutujate `scanl`, `scanr` tüübid ja saada neist aru.

337. Anda koodiga (107) defineeritud muutujale `sqrs` uus samaväärne definitsioon operaatori `scanl` abil.

338. Lahendada ülesanded 246 ja 247 operaatori `scanl` abil.

339. Asendada operaatori `iterate` argument muutuja `pasDiag` definitsioonis (157) nii, et tulemuseks oleks samaväärne definitsioon, mille korral diagonaalidele arvutuste käigus nulli ei lisataks.

340. Lahendada ülesanded 288, 289 ja 290 kõrgemat järku funktsioonide abil.

341. Anda operaatori `hanoiInter` väärtuseks funktsioon, mis võtab sisse täisarvu n ja operatsioonide jada kujul, mille produtseerib kood (120), (121) või (143), ja annab tulemuseks listi kõigi seisudega, mis nende tõstete käigus tekivad, kui algseisus on n ketast esimese varda otsas ja teised vardad on tühjad.

5.1.3 Kõrgemat järku funktsioonid protseduuriilmas

Protseduuridega seonduvaid kõrgemat järku funktsioone on eeldefineeritud päris palju. Siin kohtame neist kaht.

1. Interaktiivsed sessioonid. On olemas muutuja `interact`, mille väärtuseks on üks väga omapärane kõrgemat järku funktsioon. Ta võtab argumendiks mingi sõnesid sõnedeks teisendava funktsiooni ja annab tulemuseks protseduuri, mis kuulab standardsisendit, suunab sealt tuleva sümbolivoo funktsiooni argumentsõneks ja kirjutab funktsiooni väärtuse sel sõnel standardväljundisse.

Selline protseduur ei oleks mõttekas, kui funktsiooni argumendiks oleks vaja anda lõpuni teada olev sõne. Standardsisendit, kui ta just pole ümber suunatud, ei saa kunagi lõpuni kuulata; seega funktsiooni argument jääb üldjuhul poolikuks. Kujul `interact` f oleva avaldise väärtuseks olev protseduur loeb standardsisendit laisalt, parajasti niipalju kui teda f rakendamiseks reaalselt vaja on, ja on võimeline samal ajal kirjutama jooksvalt väljundsõnet, niipalju kui f arvutus teda on valmis saanud.

Näiteks `interact (const "Hei!\n")` väärtus on protseduur, mis kirjutab kohe ekraanile "Hei!" (koos reavahetusega) ja lõpetab normaalselt töö, ilma et oleks standardsisendist ühtki sümbolit oodanud, sest avaldise `const "Hei!\n"` väärtustamine tema argumenti ei loe.

Üldiselt kui operaator `kirjuta` on antud koodiga

```
kirjuta
  :: String -> String -> String'

kirjuta s _
  = s ++ "\n"'
```
(158)

siis iga sõnetüüpi avaldise s korral on avaldise `interact (kirjuta` s`)` väärtus selline protseduur, mis kirjutab standardväljundisse s väärtuse ja vahetab rida.

Kui `interact` argumendi `f` väärtus on selline funktsioon, mis vajab väärtuse määramiseks oma argumentsõnet täies pikkuses, siis `interact f` normaalselt tööd ka ei lõpeta.

Näiteks `interact reverse` väärtus on protseduur, mis ootab standardsisendi taga ilma nähtava reaktsioonita kasvõi lõpmatuseni, sest esimene standardväljundisse kirjutatav sümbol peaks olema standardsisendis viimane.

Interaktiivsete sessioonide esitamine kujul `interact f` vähendab protseduuritüüpe kasutavat programmiosa, sest suhtlus kodeeritakse tavalise sõnefunktsioonina. Kui programmi kogu sisend-väljund seisneb suhtluses kasutajaga, võib kogu programmi esitada kujul `interact f`. Sellisel juhul on muutuja `main` definitsioon koodi ainus koht, kus protseduuritüübid on mängus. Enamasti on selle stiili viljelemisel vaja standardsisendi ja -väljundi puhverdamine ja kaja keelata, mistõttu `main` definitsioon on kujul

```
main
  = do
      hSetBuffering stdin NoBuffering
      hSetBuffering stdout NoBuffering .   (159)
      hSetEcho stdout False
      interact f
      hSetEcho stdout True
```

Moodulis Prelude saab muutuja `id` väärtuseks samasusfunktsiooni. Võttes `f = id`, kirjutab programm (159) kõik kasutaja sisestatud sümbolid reaalajas ekraanile.

Kui `f = map toUpper`, siis toimub sama selle erinevusega, et kõik väiketähed muutuvad vastavateks suurtähtedeks. Programm kummalgi juhul küll normaalselt tööd ei lõpeta, kuid suudab pidevalt vastata kasutaja tegevusele.

Kui aga näiteks `f = take 10`, siis ootab ja kajab programm (159) esimesed 10 klahvivajutust, seejärel lõpetab normaalselt töö.

Definitsioonid operaatori `interact` kaudu võivad olla lausa tuntavalt lühemad ja elegantsemad kui otsesed.

Näiteks andes operaatori `kuulaK` koodiga

```
kuulaK
  :: String -> String'

kuulaK (c : cs@ ~(d : ds))
  = c : if c == d then d : "\n" else kuulaK cs'   (160)
```

töötab programm (159), kus ƒ = `kuulaK`, samamoodi nagu võrdlemisi pika koodiga (103) muutuja `kuulaKorduseni` väärtuseks defineeritud protseduur.

Laisk näidis koodis (160) on vajalik selleks, et ekraaniväljund püsiks klaviatuurisisendiga ühes rütmis. Kui tilde ära jätta, vajab `kuulaK` väljundi andmiseks kaht sümbolit sisendist — jooksvat ja järgmist —, mistõttu väljund oleks ühe sümboli võrra sisendist taga.

Meenutame veel programmi, mis koosnes definitsioonidest (61) ja (62) ning mis küsis kasutajalt kaks täisarvu ja vastas nende summaga. Et koodiga (159) saada samaväärne programm, võib võtta ƒ = `küsi`, mis antud koodiga

```
küsi
  :: String -> String'

küsi is
  = let
      s : ~(t : _)
        = lines is
      sum
        = read s + read t :: Integer
    in
    "Anna kaks täisarvu.\n" ++
    s ++ '\n' : t ++ '\n' :
    "Summa on  " ++ show sum ++ " .\n"
```
(161)

Ülesandeid

342. Kirjutada ƒ nii, et `interact ƒ` väärtuseks oleks sama protseduur nagu *do*-avaldisel (57).

343. Kirjutada programm kujul (159), mis kuulab standardsisendit ja kajab standardväljundisse kõik tähed, kuid mitte muid sümboleid. Programm lõpetagu töö, kui vajutatakse *escape*-klahvi.

344. Anda ülesannetele 170 ja 173 uued lahendused kujul (159).

345. Kirjutada ülesannetele 211, 212, 215, 256 uued lahendused, kus kogu interaktiivne osa oleks pakitud ühte operaatori `interact` argumenti.

346. Kirjutada definitsioonidega (81), (82), (83), (84) kirjutatud klaviatuuril tippimise harjutamise programm ümber nii, et kogu interaktiivsus oleks pakitud operaatori `interact` argumenti ühel väljakutsel.

347. Miks on koodis (161) lokaalsete muutujate `s` ja `t` definitsioonis tilde oluline?

2. Erindite töötlemine. Kõrgemat järku funktsioone läheb vaja ka erindite püüdmiseks.

2.1. Erindi püüdmine. Selleks otstarbeks pakub moodul Prelude operaatori `catch`. Tema väärtuseks on karritatud funktsioon, mis võtab argumentideks protseduuri a ja funktsiooni h, mis igale erindile seab vastavusse a-ga sama tüüpi protseduuri, ja annab tulemuseks protseduuri, mis täidab kõigepealt a: kui see lõpetab normaalselt, siis muud ei juhtugi ja edastatakse protseduuri a edastusväärtus; kui aga a heidab erindi e, siis täidetakse lisaks protseduur $h\,e$, misjärel sõltuvalt tema töö kulust kas edastatakse tema edastusväärtus või heidetakse tema heidetud erind.

Funktsiooni h roll niisiis on pakkuda alternatiivne tegevus igaks erandjuhtumiks. Sellist funktsiooni nimetatakse erindipüüniseks.

Operaator `catch` seega võimaldab hargnemist vastavalt sellele, kas protseduuri täitmine kulgeb normaalselt või heidab ta erindi. Teiseks argumendiks oleva püünise väljunditüüp peab võrduma esimeseks argumendiks oleva protseduuri tüübiga samal põhjusel, miks tavalistes hargnemiskonstruktsioonides peavad harude tüübid kokku langema.

Näiteks võiksime oma faililugemiskoodi (59) täiendada deklaratsioonidega

```
lugemineE
  :: IO ()'

lugemineE
  = lugemine `catch`                              .
    const (putStrLn "Lugemisel tekkis erind!")
```

Kutsudes nüüd välja muutuja `lugemineE`, toimub õnnestunud failiavamise puhul kõik nagu varem, ebaõnnestumise puhul aga ilmub pärast failinime sisestamist ekraanile kiri “Lugemisel tekkis erind!”. Veateadet ei teki.

2.2. Erindite liigid. Tavaliselt soovib programmi kasutaja erindi tekkimisel saada täpsemat infot erindi iseloomu kohta.

Näiteks faili lugemise puhul on kaks põhimõtteliselt erinevat erindi tekke võimalust: faili puudumine ja tema lugemisõiguse puudumine.

Moodulist System . IO . Error tulevad muutujad, mille väärtuseks on predikaadid, mis näitavad erindi iseloomu. Kasutades neid, võime kirjeldada püüniseid, mille töö sõltub erindist.

Faililugemise näites sobib kirjutada

```
püünis
  :: IOError -> IO ()'

püünis e
  | isDoesNotExistError e
    = putStrLn "Pole faili!"
  | isPermissionError e                    .     (162)
    = putStrLn "Pole faili lugemisõigust!"
  | otherwise
    = putStrLn "Tekkis tundmatu erind."
```

Erinditöötlusega lugemise protseduuri kirjeldab nüüd kood

```
lugemineE                                        (163)
  = lugemine `catch` püünis
```

Tihti on vaja erinevaid erindeid töödelda erinevas kohas erinevate püünistega. Püünised peaksid siis erindeid valikuliselt töötlema: osa kinni püüdma, teised läbi laskma, et nad õigesse püünisesse jõuaksid.

Erindi läbilaskmiseks peab püünis sama erindi uuesti heitma. Seda võib saavutada tuttava operaatoriga `ioError`. Erinditöötluse vaatepunktist on `ioError` väärtus püünis, mis ühtki erindit kinni ei pea.

Asendame faililugemisprogrammis püünise kahe erinevaga, millest üks püüab ainult õiguste puudumisest, teine aga ainult faili puudumisest tekkivaid erindeid.

Olgu kavas näiteks õigustega seotud erindite püünis asetada enne faili puudumise erindite püünist. Sellele vastab muutuja `lugemineE` definitsioon

```
lugemineE
  = lugemine `catch` püüaÕigused `catch` püüaMuud'
```

kus `püüaÕigused` ja `püüaMuud` tuleb sobivalt defineerida.

Et õigustega seotud erindite püünis on enne teist, peab ta teise püünisesse mõeldud erindid läbi laskma ehk uuesti heitma. Sobib kood

```
püüaÕigused
  :: IOError -> IO ()'

püüaÕigused e
  | isPermissionError e
    = putStrLn "Pole faili lugemisõigust!" .
  | otherwise
    = ioError e
```

Muutuja `püüaMuud` jaoks saame

```
püüaMuud
  :: IOError -> IO ()'

püüaMuud e
  | isDoesNotExistError e
    = putStrLn "Pole faili!"           .
  | otherwise
    = putStrLn "Tekkis tundmatu erind."
```

Ülesandeid

348. Lasta interaktiivsel keskkonnal kuvada muutuja `catch` tüüp ja saada sellest aru.

349. Kirjutada programm, mis küsib käivitudes kasutajalt faili nime ja kirjutab saadud nimega faili sisu ekraanile. Kui faili lugemine ebaõnnestub, siis teatab veast ja alustab sama tööd otsast peale.

350. Ülesande 213 lahendusena saadi klaviatuuril tippimise harjutamise programm, kus harjutamiseks esitatavaid ridu loeti failist. Kirjutada seal muutuja `harjutus` definitsioon ümber nii, et kui nõutud faili lugemisel tekib tõrge, siis programm katkestab harjutuse ja teatab eesti keeles põhjusest.

351. Kirjutada ülesande 350 lahenduses muutujate `harjutus` ja `kordaTähed` definitsioonid ümber nii, et *escape*-klahvi vajutamisel lõpetaks harjutamisprogramm kohe normaalselt töö.

5.1.4 Kõrgemat järku funktsioonide defineerimine

1. Süntaks. Kõrgemat järku funktsioonide definitsioonidel pole mingeid erilisi väliseid tunnuseid. Nad kasutavad samu süntaktilisi konstruktsioone.

Et konstruktorid peavad näidises olema täisargumenteeritud, siis funktsioonitüüpi argumendi näidiseid konstruktoriga moodustada ei saa. Mõttekad on ainult muutuja ja jokker (ehknäidis ja laisk näidis ei oma ilma konstruktorita praktilist sisu). Muutuja peab argumendinäidises olema prefikskujul.

Tahame näiteks saada muutuja `loenda` väärtuseks kõrgemat järku funktsiooni, mis võtab argumendiks predikaadi ja seejärel listi ning annab tulemuseks predikaati rahuldavate elementide arvu listis. Sellise operaatori signatuur oleks

```
loenda
  :: (a -> Bool) -> [a] -> Int
```

Esmase definitsiooni kirjutamiseks märkame, et vajalik tegevus sarnaneb operaatori `filter` tööga, kuid vaja on sõelale jäänud elemendid loendada, mitte neist listi koostada. Et aga listi elementide arvu jaoks on olemas operaator `length`, saame eeldefineeritud operaatorite abil hõlpsasti sobiva definitsiooni

```
loenda p xs
  = length (filter p xs)
```
(164)

Muutuja `p` märgib argumentpredikaati, mis, nagu näha, antakse kohe operaatorile `filter` edasi.

Nüüd näiteks on korrektne avaldis `loenda (< 5) [1 .. 100]`; tema väärtus on 4, sest 5-st väiksemad arvud 1-st 100-ni on 1, 2, 3, 4, mida on kokku 4. Samuti on korrektne näiteks avaldis `loenda isUpper "Tere, Haskell!"`, mille väärtus on 2 — suurtähtede arv tekstis "Tere, Haskell!".

Uut operaatorit `loenda` saab ära kasutada ka näiteks defineerimaks operaatorit `nullideArv`, millele oleme andnud varem definitsioonid (72) ja (98). Selleks tuleb predikaadiks valida selline, mis on tõene parajasti nullil; sobib kood

```
nullideArv xs
  = loenda (0 ==) xs
```
(165)

Niisama lihtne on sisse tuua üldisem operaator `loendaElem`, mille korral loendatakse nullide asemel mistahes etteantud võrdusega tüüpi andme esinemisi, koodiga

```
loendaElem x xs
  = loenda (x ==) xs;
```
(166)

signatuur on

```
loendaElem
  :: (Eq a)          .
  => a -> [a] -> Int
```

Teise näitena võib defineerida operaatori, mis võimaldaks interaktiivseid sessioone välja kutsuda lihtsamalt kui programmiga kujul (159). Operaator peaks võtma funktsioonitüüpi avaldisena esitatud sessiooni argumendiks ja panema ta vormis (159) ƒ kohale. Sobib definitsioon

```
täida f
  = do
      hSetBuffering stdin NoBuffering
      hSetBuffering stdout NoBuffering ,
      hSetEcho stdout False
      interact f
      hSetEcho stdout True
```
(167)

signatuur on

```
taida
  :: (String -> String) -> IO ()
```

Nüüd piisab koodinäidete (160) ja (161) testimiseks väärtustada interaktiivses keskkonnas avaldised `täida kuulaK` ja `täida küsi`.

Ülesandeid

352. Lahendada ülesanded 343 ja 344, kasutades programmikuju (159) asemel koodiga (167) defineeritud muutujat `täida`.

2. Argumendi kasutamine funktsioonina. Kui defineeritava operaatori argument on funktsioonitüüpi, siis võib teda märkivat muutujat vajadusel kasutada ka otseseks rakendamiseks. Lubatud on nii prefiks- kui infikskuju.

Näites (164) anti funktsioonitüüpi formaalne parameeter vaid teisele operaatorile (`filter`) edasi.

On aga võimalikud definitsioonid nagu näiteks

```
kõrgem f
  = f 1
```
(168)

Siin muutuja `f`, mis vasakul on argumendipositsioonis, on paremal ise argumendile rakendatud. Muutuja `kõrgem` väärtuseks saab kõrgemat järku funktsioon, mis võtab argumendiks funktsiooni ja annab välja selle funktsiooni väärtuse kohal 1.

Tüüpide kohta näitab see, et argumentfunktsioon peab olema arvudele rakendatav. Operaatori `kõrgem` signatuuriks sobib

```
kõrgem
  :: (Num a)
  => (a -> b) -> b
```

Tüübimuutuja `b` esineb kaks korda, sest `kõrgem` väärtuseks olev funktsioon annab välja andme, mille annab välja tema argumentfunktsioon, st tema väärtusetüüp ja argumentfunktsiooni väärtusetüüp langevad kokku.

Praktilisemate näidete saamiseks märkame, et `loenda` definitsioon (164) pole arvutuslikult parim, sest tema järgi arvutamisel tekib vahestruktuur. Võrdluseks, operaatori `nullideArv` definitsioonid (72) ja (98) ei tekita vahestruktuuri.

Seepärast kirjutame ka operaatorile `loenda` uued, rekursiooniga definitsioonid, mis arvutusskeemi poolest vastavad `nullideArv` definitsioonidele (72) (otsene rekursioon) ja (98) (akumulaatoriga).

Alustame definitsiooniga (72); lisades predikaati märkiva parameetri ja asendades nulliga võrdumise selle parameetri rakendamisega, saame koodi

```
loenda p (x : xs)
  = let
      ülejäänu
        = loenda p xs
    in
    if p x then 1 + ülejäänu else ülejäänu
loenda _ _
  = 0
```
(169)

Koodi (98) samamoodi ümber tehes aga saame definitsiooni

```
loenda p xs
  = let
      arv n (z : zs)
        = arv (if p z then n + 1 else n) zs
      arv n _
        = n
    in
    arv 0 xs
```
. (170)

Hoolimata välise operaatori lisandunud parameetrist `p` on lokaalse operaatori `arv` argumendid samad mis eeskujuks olnud definitsioonis. Funktsiooniargumenti pole vaja lokaalses definitsioonis kaasas tassida, sest arvutuse käigus ta väärtus ei muutu.

Veel ühe lihtsa näitena vaatame, kuidas defineerida `map`-laadne operaator `map2`, mille väärtuseks oleks funktsioon, mis võtab argumendiks kaks funktsiooni ja listi ning rakendab argumentfunktsioone listi elementidele vaheldumisi. Signatuur oleks

```
map2
  :: (a -> b) -> (a -> b) -> [a] -> [b]'
```

mis näitab, et argumentfunktsioonid on sama tüüpi. See on vältimatu, sest mõlemad funktsioonid peavad olema rakendatavad sama listi elementidele ning ka tulemuslisti jäävad vaheldumisi kummagi funktsiooniga tekitatud elemendid — listi kõik elemendid peavad aga olema sama tüüpi.

Definitsiooniks sobib

```
map2 f g (x : xs)
  = f x : map2 g f xs
map2 _ _ _
  = []
```
.

Defineeritud operaatorit saab kasutada mitmeks otstarbeks, ka siis, kui üle ühe tuleb elemendid muutmata jätta.

Näiteks olukorras, kus arvulisti iga teise elemendi kohale tuleb võtta tema vastandarv, võime `map2` teiseks argumendiks panna `(* (-1))`, esimeseks argumendiks aga `(* 1)`. Triviaalsete korrutamiste asemel saab ka intelligentsemalt: vastandarvu leidmine on eeldefineeritud muutuja `negate` väärtuseks ning samasusteisendus muutuja `id` väärtuseks. Niisiis avaldise `map2 id negate [0 .. 4]` väärtus on lõplik list elementidega $0, -1, 2, -3, 4$.

Avaldise `map2 toUpper toLower "kera"` väärtus on aga "KeRa" ning sama väärtuse saame ka siis, kui sõneargumendi kohal on `"kera"` asemel näiteks `"Kera"` või `"KERA"`.

Tekib küsimus, kas muutujat `map2` ei saanuks samaväärselt defineerida kähku, kasutades vaid eeldefineeritud operaatoreid, nii nagu kood (164) defineeris muutuja `loenda`? Aga palun, elegantne definitsioon on

```
map2 f g xs
  = zipWith ($) (cycle [f, g]) xs .
```
(171)

Ülesandeid

353. Otsida Hugsi teegist mooduli Prelude algtekstist üles operaatorite `$`, `flip`, `curry`, `uncurry`, `.`, `iterate`, `until`, `takeWhile`, `dropWhile`, `filter`, `map`, `zipWith`, `foldr`, `foldl`, `scanl` definitsioonid ja saada neist aru.

354. Defineerida muutuja `?` väärtuseks postfiksne funktsioonirakendamine (st vastandina operaatorile `$`, peab ta võtma funktsiooni paremaks ja argumendi vasakuks argumendiks).

355. Olgu muutuja `kõrgem` defineeritud koodiga (168). Milliseid avaldistest

    `log 5.0`, `log`, `(5.0 -)`, `(-)`, `properFraction`, `fst`, `(: [88])`, `take`, `takeWhile`, `foldr (+)`, `foldr`

    saab tüübikorrektselt anda `kõrgem` argumendiks? Mis on `kõrgem` rakendamisel saadavate avaldiste väärtus neil argumentidel?

356. Kasutades koodiga (164), (169) või (170) defineeritud operaatorit `loenda`, kirjutada interaktiivses keskkonnas avaldis, mille väärtus on funktsioon, mis võtab argumendiks funktsioonide listi ja annab välja arvu, mitu funktsiooni selles listis annavad 0-le rakendades tulemuseks 1.

357. Defineerida muutuja `filterPalju` väärtuseks funktsioon, mis võtab argumendiks järjest predikaatide listi ja teise listi, mille elementitüüp võrdub predikaatide argumenditüübiga, ja annab tulemuseks listi, mille elementideks on

kõik need listid, mis saadakse, kui predikaatide listis fikseeritakse üks predikaat ja jäetakse teises argumentlistis järele ainult seda rahuldavad elemendid.

358. Defineerida muutuja `järgFilter` väärtuseks funktsioon, mis võtab järjest argumendiks predikaadi p ja listi l ning annab välja listi neist listi l elementidest järjekorda muutmata, mis l-s vahetult järgnevad mõnele predikaati p rahuldavale elemendile. Testida erisuguste argumentpredikaatidega.

359. Defineerida muutuja `eemaldaSellineEt` väärtuseks funktsioon, mis võtab argumendiks predikaadi ja listi, mille elemendituup võrdub predikaadi argumendituubiga, ja annab tulemuseks paari, mille esimene komponent on True või False vastavalt sellele, kas predikaati rahuldavaid elemente listis leidub või mitte, ja teine on list, mis saadakse argumentlistist esimese predikaati rahuldava elemendi väljajätmisel (kui selliseid pole, siis argumentlist ise).

    Anda ülesandes 190 nõutud operaatorile `eemaldaEsim` uus definitsioon selle muutuja kaudu.

360. Defineerida muutuja `binAll` väärtuseks kõrgemat järku funktsioon, mis võtab argumendiks binaarse predikaadi p ja annab välja funktsiooni, mis võtab argumendiks listi ja annab välja True või False vastavalt sellele, kas list on lõplik ja p mistahes kahest järjestikusest elemendist on tõene või leidub listis koht, kus p kahest järjestikusest elemendist on väär.

    Operaatorit `binAll` kasutades kirjutada avaldisi, mis kontrollivad mingite listide kohta, kas nad on kasvavalt järjestatud, mittekahanevalt järjestatud, kahanevalt järjestatud, mittekasvavalt järjestatud, vahelduvate märkidega.

361. Olgu muutuja `loeDropWhile` defineeritud koodiga

```
loeDropWhile p xs
  = (dropWhile p xs , length (takeWhile p xs))
```

    Anda sellele muutujale samaväärne definitsioon otsese rekursiooniga.

362. Defineerida muutuja `värskenda` väärtuseks karritatud funktsioon, mis võtab argumentideks funktsiooni f, listi l ja arvu i ning annab tulemuseks listi, mille saab listist l tema i-ndale elemendile (lugemine alates 0-st) funktsiooni f rakendamisel.

363. Defineerida muutuja `mapUntil` väärtuseks karritatud funktsioon, mis võtab argumentideks predikaadi p, funktsiooni f ja listi l ning annab tulemuseks listi, mille saab listist l, kui rakendab järjest igale elemendile funktsiooni f kuni esimese selliseni, mis annab tulemuseks tingimust p rahuldava väärtuse (viimane kaasaarvatud), järelejääv listisaba jääb tulemuslisti lõppu.

364. Defineerida muutuja `mapIf` väärtuseks funktsioon, mis võtab argumentideks predikaadi p, funktsiooni f ja listi l, mille elemenditüüp langeb kokku nii p kui ka f argumenditüübiga, ja annab tulemuseks listi, mille saab l-st kõigile p-d rahuldavatele elementidele f rakendamisega.

365. Kui definitsiooni (171) paremas pooles panna `zipWith` argumendi `($)` asemele `id`, kas tekkiv kood on (a) tüübikorrektne? (b) endisega samaväärne?

5.2 Argumendivaba stiil

Oleme juba näinud, et sama funktsioonitüüpi muutuja võib olla eri kordadel kasutatud erineva arvu argumentidega. Muuhulgas võib argumentide arv muutuja kasutuses olla erinev argumentide arvust tema definitsioonis. Ainsa piirangu argumentide arvule panevad vaid tüübid: avaldisel, mis pole funktsioonitüüpi, ei saa olla argumenti.

Näiteks teame, et `isUpper` on tavaliselt rakendatud sümbolitüüpi argumendile. Kuid korrektses avaldises `filter isUpper "Tere, Haskell!"` tal argument puudub. Operaatorit `flip` võime kasutada kolme argumendiga (funktsioon ja tema kaks argumenti), kuid avaldises `uncurry (flip div)` on tal ainult üks.

Operaatori `püünis` definitsioonis (162) on tal argument, kuid kasutavas koodis (163) on ta ilma argumendita.

Operaatorit `const` sai kasutada lausa kuitahes paljude argumentidega.

Muutuja võib olla täisargumenteerimata ka oma definitsioonis — teda defineeriva deklaratsiooni vasakus pooles võib olla vähem argumente kui oleks minimaalselt vaja tema täisargumenteerimiseks. Selline definitsioon öeldakse olevat argumendivabas stiilis.

See tähendab funktsiooni kirjeldamist otse, selle asemel et kirjeldada tema väärtusi argumentidel. Argumendivaba stiil on väga laialt kasutatav nii funktsionaalses programmeerimises kui ka matemaatilistes distsipliinides.

5.2.1 Ekstensionaalsus

Argumendivaba stiili aluseks on ekstensionaalsus — matemaatiliste funktsioonide fundamentaalne omadus, mille kohaselt funktsioonid on võrdsed, kui nad annavad samadel argumentidel sama tulemuse. Matemaatiline

funktsioon on puhas seos argumentide ja väärtuste vahel, ta ei hõlma mitte mingit lisateavet nagu nimi, identiteet vms.

1. Argumendivabale stiilile ülemineku põhisamm. Ekstensionaalsus võimaldab selle asemel, et kirjutada $f\,x = g\,x$ iga võimaliku argumendi x jaoks, kirjutada lihtsalt $f = g$. Samal põhimõttel võib argumente ära jätta Haskelli definitsioonidest.

1.1. Argumendi taandamine. Kui defineeritava funktsiooni töö seisneb lihtsalt oma argumendi ettesöötmises mingile teisele funktsioonile, võib selle argumendi mõlemalt poolt võrdusmärki ära "taandada".

Vahetu näite saame, kui vaatleme põimemeetodil järjestamise abioperaatori `jupid` definitsiooni (129), mis ütleb, et suvalise listi "jupid" on tema segmendid. Jättes mõlemalt poolt ära argumendi `xs`, saame samaväärse definitsiooni

```
jupid
  = segmendid.
```
(172)

Mõte on nii või teisiti see, et muutujate `jupid` ja `segmendid` väärtuseks on sama funktsioon, ning kood (172) väljendab seda mõneti otsesemalt kui varasem.

Samuti saab taandada ühise argumendi `xs` muutuja `nullideArv` definitsioonist (165), saades koodi

```
nullideArv
  = loenda (0 ==).
```

Üleminekul argumendivabale stiilile võib tekkida vajadus lisada signatuur.

Kui koodis (129) pole signatuur kohustuslik, siis definitsioon (172) ilma signatuurita annab tüübivea.

Paremas pooles tuleb argumendi väljatoomiseks vajadusel infiksne rakendamine asendada prefikssega.

Funktsioonitasemele saab viia ka lõigu poolitamise meetodi juures kasutatud abioperaatori `algpiirid` definitsiooni (118). Selleks kirjutame kõigepealt paari paremal pool ümber prefiksse rakendamisega, saades definitsiooni

```
algpiirid n
  = (,) 1 n'
```

sellest aga saame juba tuttaval viisil argumendivabas stiilis definitsiooni

```
algpiirid
  = (,) 1.
```
(173)

1.2. Taandamise lubatuse kriteerium. Ekstensionaalsus lubab ühist argumenti vasaku ja parema poole lõpust kaotada parajasti siis, kui see argument ei esine definitsioonis mujal. Kui definitsiooni parem pool sisaldaks seda muutujat veel, sõltuks ka rakenduv funktsioon sellest argumendist. Sellist olukorda ekstensionaalsusseadus ei kata.

Sellises olukorras argumendi taandamisel jääks see muutuja paremasse poolde kas defineerimata (tulemus poleks süntaktiliseltki korrektne) või hoopis kusagil mujal defineeritud muus tähenduses.

Näiteks operaatori `sqr` definitsioon (37) omandab prefikssele rakendamisele ülemineku samaväärse kuju

```
sqr x
  = (*) x x.
```

Selles aga mõlemalt poolt argumenti `x` kaotada ei saa, sest paremal esineb ta kaks korda. Esimene esinemine paremas pooles jääks sinna defineerimata rippuma.

Kui alustada samaväärsest, kuid ümbernimetatud lokaalse muutujaga definitsioonist

```
sqr sin
  = (*) sin sin'
```

siis vigane taandamine jätaks muutuja `sin` paremasse poolde siinuse tähenduses.

Ülesandeid

366. Lahendada ülesanne 312 argumendivabas stiilis.

367. Anda koodiga (111) defineeritud muutujale `kordusteta` samaväärne definitsioon argumendivabas stiilis.

2. Mõned erijuhud.

2.1. Mitme argumendiga definitsioonid. Definitsiooni pooltele ühiseid argumente, mis mujal ei esine, võib lõpust ära jätta ka juhul, kui need argumendid pole ainsad. See on tavaline ekstensionaalsus karritatud funktsioonidel. Kui definitsiooni poolte lõpus on mitu ühist argumenti samas järjestuses, siis võib nad kõik ära jätta, sest niisama hästi saaks nad kaotada ühekaupa.

Samuti nagu muutuja `nullideArv` definitsioonis (165), saab taandada argumendi `xs` järgmisest, muutuja `loendaElem` definitsioonist (166), saades koodi

```
loendaElem x
  = loenda (x ==)'
```
(174)

kus siiski on vasakule poole üks argument järele jäänud.

2.2. Hargnemisega definitsioonid. Ekstensionaalsust saab kasutada ka hargnemisega definitsioonis. Kui iga haru sisu moodustab millegi rakendamine vasakul viimase argumendina esinevale muutujale ning need on ainsad selle muutuja esinemised selles tähenduses, siis tohib taandada selle muutuja korraga nii vasakult kui ka igast harust. Konkreetne hargnemiskonstruktsioon tähtsust ei oma.

Lihtsa näitena defineerime funktsiooni, mis võtab argumendiks järjest tõeväärtuse b, kaks ühte tüüpi funktsiooni f ja g ning mõlemale argumendiks sobiva andme x ning vastavalt tõeväärtusele b annab tulemuseks kas f või g rakendamise tulemuse x-le. Signatuur oleks

```
rakEmbKumb
  :: Bool -> (a -> b) -> (a -> b) -> a -> b.
```

Tavastiilis definitsioon võib siin välja näha kujul

```
rakEmbKumb b f g x
  = if b then f x else g x
```
(175)

ning vastav argumendivabas stiilis definitsioon oleks

```
rakEmbKumb b f g
  = if b then f else g.
```

2.3. Tüübitaseme ekstensionaalsus. Argumente võib samamoodi kaotada ka tüübisünonüümide definitsioonides.

Näiteks deklaratsiooni

```
type Proov a b
  = Either (Maybe a) b
```

asemel võib samaväärselt kirjutada

```
type Proov a
  = Either (Maybe a).
```

Tüübisünonüümide kasutus väljaspool oma definitsiooni peab siiski jääma täisargumenteerituks.

Deklaratsioon

```
type Flip f a b
  = f b a
```

on küll legaalne, kuid temast pole kasu argumentide kaotamisel muudest definitsioonidest. Näiteks

```
type UusProov b
  = Flip Either (Maybe b)
```

ei ole lubatud, sest tüübisünonüüm `Flip` pole täisargumenteeritud.

Ülesandeid

368. Otsida Hugsi teegist üles muutuja `subtract` definitsioon ja saada temast aru.

369. Lahendada ülesanne 316 argumendivabas stiilis.

370. Kirjutada muutuja `rakEmbKumb` definitsioon (175) samaväärselt ümber valvuritega ja minna sealt argumendivabale stiilile. Teha sama ka deklaratsioonisüsteemiga.

371. Argumendivabas stiilis defineerida tüübisünonüümi `KahPaar` väärtuseks tüübifunktsioon, mis argumendil A annab tulemuseks tõeväärtusetüübist tüüpi A töötavate funktsioonide tüübi.

3. Argumendivaba stiili piirid. Õpiku alguses sai möödaminnes mainitud, et Haskelli maailmas ekstensionaalsus rangelt võttes alati ei kehti. Nimelt funktsioonitüübi bottom ja sama tüüpi tõeline funktsioon, mis igal argumendil annab välja väärtusetüübi bottomi, annavad rakendamisel suvalisele argumendile sama väärtuse (bottomi). Samas nad siiski on ilmutatud viidatavuse põhjal erinevad, milles võib veenduda neid endid mõnele agarale kõrgemat järku funktsioonile argumendiks andes.

Annoteeritud avaldise `undefined :: Int -> Int` väärtus on täisarvudel töötavate funktsioonide tüübi bottom. Kuid tema rakendamine argumentidele tüübist Int on tüübikorrektne, seega võib seda teha. Kuna sellisel rakendamisel lõpetab arvutus töö veateatega, mille tekitab `undefined`, on rakendamise tulemus bottom.

Kirjutame ka signatuuri ja definitsiooni

```
libabottom
  :: Int -> Int'

libabottom n
  = undefined n'
```
(176)

Siis `libabottom` väärtuseks saab (tõeline) funktsioon, mis annab samuti konstantselt välja bottomi.

Seejuures lõpeb `const 0 $! (undefined :: Int -> Int)` väärtustamine veaga, samas annab `const 0 $! libabottom` väärtustamine vastuse `0`.

Seega definitsioonis (176) argumendi kaotamine muudab defineeritava muutuja väärtust. Kui see ei sobi, siis argumendivaba stiili kasutada ei saa.

Kuna kirjeldatud ekstensionaalse rikkumised on perifeerset laadi, võib ekstensionaalsuse lugeda Haskellis siiski praktiliselt kehtivaks.

5.2.2 Metoodiline üleminek argumendivabale stiilile

Argumendivaba stiili rakendamiseks võib olla vaja algset definitsiooni teisendada, et parem pool viia nõutavale kujule.

Parema poole lihtsat mudimist nägime juba muutuja `algpiirid` defineerimisel koodiga (173).

Tüüpiline võte on mitmesuguste kõrgemat järku funktsioonide sissetoomine, et esitada parem pool millegi rakendamisena vajalikule argumendile. Arvestagem, et seda võtet ohjeldamatult viljeldes võib koodi ka lausa loetamatuks muuta. See võte on kui vürts, olles soovitav vaid väikeses koguses.

1. Esitused ümberpaigutavate funktsioonide kaudu.

1.1. Üleminek argumentide vahetamisega. Lihtsamal juhul piisab argumentide järjekorra muutmisest, et osast argumentidest vabaneda.

Vaatame järjestamise kiirmeetodi realisatsiooni akumulaatoritehnikaga, operaatori `järjestaKiir` definitsiooni (122). Defineeritava muutuja ainus argument `xs` esineb küll paremas pooles — võtmesõna **`in`** järel avaldises `kiir xs []` —, kuid et ta pole viimane argument, ei ole tema vahetu taandamine võimalik.

Kuna selles kohas on rakendatavaks operaatoriks sama definitsiooni lokaalne operaator `kiir`, on üks lahendusvõimalus lokaalses definitsioonis argumentide järjekord vahetada. Siis vahetuvad ühtlasi argumendid ka väljakutsel, `xs` saab viimaseks ning üleminek argumendivabale stiilile võimalikuks.

See lahendus aga pole kõige parem mitmel põhjusel. Esiteks halvendaks argumentide järjekorra vahetamine lokaalse operaatori definitsiooni loetavust. Akumulaatoriga definitsioonid on tavaliselt kõige paremini jälgitavad just siis, kui akumulaator on viimane argument, sest siis satuvad järjestikku toimuvad tegevused oma õiges järjekorras ka definitsiooni. Teiseks on antud juhul võimalik see akumulaator ise argumendivaba stiili abil kergesti definitsioonist kaotada, nagu näeme edaspidi, selleks aga peab ta olema viimane.

Seepärast läheneme olukorrale teisiti: kirjutame avaldise `kiir xs []` samaväärselt ümber kujul `flip kiir [] xs`. Selles avaldises on `xs` viimane argument, nii et tee argumendivabale stiilile on avatud. Argumendi taandamisel saame koodi

```
järjestaKiir
  = let
      kiir (x : xs) as
        = let
            (us , vs)
              = kaheksLahuta x xs
          in
            kiir us (x : kiir vs as)
      kiir _        as
        = as
    in
    flip kiir []
```
. (177)

Ülesandeid

372. Defineerida argumendivabas stiilis muutuja `jäägid` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab välja listi, mille elementideks on suurusjärjestuses kõik mittenegatiivsed n-st väiksemad täisarvud.

373. Defineerda võimalikult lühidalt muutuja `flipInfiks` väärtuseks karritatud funktsioon, mis võtab järjest kolm argumenti x, f, y, millest teine on ise karritatud funktsioon, ja annab neil tulemuseks $f\,y\,x$. (Kui f kohal on infiksoperaator $\oplus$, siis on mõneti tegemist infiksoperaatori argumentide vahetamisega, sest tulemuseks on $y \oplus x$.)

1.2. Üleminek kompositsiooni abil. Väga tihti on võimalik argumentidest vabaneda kompositsiooni sissetoomisega.

Läheme argumendivabale stiilile üle ka koodi (177) lokaalses definitsioonis.

Esimese deklaratsiooni parem pool kirjeldab, mida akumulaatoriga tehakse. Talle rakendub `kiir vs`, selle rakendamise ees on `x :`, mida võib väljendada vastava sektsiooni rakendamisena, ja tulemusele omakorda rakendub `kiir us`. Kirjutades järjestikused rakendamised ümber kompositsiooni abil, tekib avaldis `(kiir us . (x :) . kiir vs) as`.

Teise deklaratsiooni paremaks pooleks on lihtsalt `as`. Selle teeme samaväärselt rakendamiseks ümber, lisades operaatori `id`, mille väärtus on samasusfunktsioon.

Tulemusena on igas harus (deklaratsioonisüsteemi deklaratsioonis) `as` kogu muu osa argument, mis võimaldab `as` kõikjalt ära koristada. Saame deklaratsiooni

```
järjestaKiir
  = let
      kiir (x : xs)
        = let
            (us , vs)
              = kaheksLahuta x xs
          in
          kiir us . (x :) . kiir vs
      kiir _
        = id
    in
    flip kiir []
```
. (178)

Kui algse definitsiooni (122) kirjutasime loetavust ohvriks tuues akumulaatori abil, et vältida tühiümbertõstmisi, siis argumendivabas stiilis definitsioonis (178) õnnestus ühendada väljenduse selgus ja arvutuse efektiivsus. Viimases ütleb lokaalse muutuja `kiir` definitsioon tähelepanu detailidele viimata otse ära järjestamise kiirmeetodi põhimõtte: lahutada listi saba elemendid suuruse järgi kahte ossa, võttes lahkmeks esimese elemendi, ja panna tulemusse algul samaviisi järjestatud esimene osa, selle järele algne esimene element ja lõpuks samaviisi järjestatud teine osa.

Teiseks vaatame uuesti operaatorit `loendaElem`, mille definitsioonist (166) õnnestus argumendivaba stiili kasutades ära kaotada üks argument kahest ja saada lühem definitsioon (174). Osutub, et pole probleem ka teine argument kaotada. Kirjutades sektsiooni ümber tavalise rakendamisega, teisendub definitsioon (174) kujule

```
loendaElem x
  = loenda ((==) x).
```

Nüüd on aga kohe näha paremas pooles olev järjestrakendamine, mistõttu on saadud definitsioon samaväärne definitsiooniga

```
loendaElem
  = loenda . (==).
```
(179)

Võrreldes definitsiooniga (174) definitsioon (179) vaevalt et loetavam on.

Ülesandeid

374. Otsida Hugsi teegist üles muutujate `odd`, `any`, `all`, `elem` definitsioonid ja saada neist aru.

375. Kirjutada ülesandele 241 uus lahendus, mis kasutaks argumendivaba stiili nii põhioperaatori kui ka abioperaatori definitsioonis.

376. Lahendada ülesanne 322, kasutades argumendivaba stiili.

377. Anda ülesandes 139 defineeritud muutujale `takeLõpust` uus samaväärne definitsioon, milles ilmutatult esineb vaid üks argument.

378. Kirjutada muutuja `suurSumma` definitsioon (105) samaväärselt ümber nii, et lokaalse operaatori definitsioonis oleks kasutatud argumendivaba stiili.

379. Lahendada ülesanne 378, kui aluseks on definitsiooni (105) asemel definitsioon (114), st arvutamisel peab akumulaatoris toimuma jooksev väärtustamine.

380. Lahendada ülesanne 316 argumendivabas stiilis nii, et ükski argument poleks ilmutatud.

381. Anda ülesandes 139 defineeritud muutujale `takeLõpust` selline samaväärne definitsioon, milles ükski argument pole ilmutatud.

1.3. Üleminek argumentide kokkuvõtmisega. Mõnikord õnnestub argumendivabale stiilile üle minna, kui kaks argumenti operaatoriga `uncurry` ühte paari kokku võtta ja vajadusel teisal operaatoriga `curry` uuesti lahutada.

Vaatame muutuja `loenda` esimest definitsiooni (164). Paremas pooles on avaldis `length (filter p xs)`, kus operaatori `loenda` parameetritele `p` ja `xs` rakendatakse algul operaatorit `filter` ja seejärel tulemusele operaatorit `length`.

Ometi ei saa seda kompositsiooni abil otse ümber kirjutada, sest rangelt võttes saab `filter`, mille väärtus on karritatud, neist kahest parameetrist argumendiks ainult esimese, `p`. Lihtsal viisil saaks kaotada vaid teise argumendi, minnes üle kompositsioonile `length . filter p`. Saadud definitsioonis saaks mõningase vaevaga ka argumendi `p` kaotada, kuid tulemus oleks juba raskesti loetav.

Alternatiivne variant on argumendid kokku võtta. Kuna algselt söödetakse nad operaatorile `filter`, tuleb just temale rakendada operaator `uncurry`. Siis on koostöö operaatoriga `length` tavaline järjestrakendamine, mida väljendab avaldis

`length . uncurry filter`. Et aga tulemuseks on vaja saada siiski karritatud funktsioon, peame selle järjestrakendamise omakorda karritama. Saame koodi

```
loenda
  = curry (length . uncurry filter)
```

Ülesandeid

382. Kirjutada argumendivabas stiilis samaväärselt ümber kahe arvu aritmeetilise keskmise definitsioon (38).

383. Kirjutada argumendivabas stiilis samaväärselt ümber definitsioon

```
tõstaLõppu n xs
  = let
      (us , vs)
        = splitAt n xs
    in
    vs ++ us
```

2. Esitused kokkuarvutavate funktsioonide kaudu. Vaatame siin selliste argumendivabas stiilis definitsioonide koostamist, kus funktsioon väljendatakse kas `foldr` või `foldl` rakendamisega kahele argumendile.

Operaator `foldr` abstraheerib kõiki selliseid rekursiooniskeeme listidel, kus tulemus mittetühjal listil leitakse listi pea ja sama arvutuse tulemuse järgi listi sabal. See tuleb välja esitustest (152) ja (153), mis avaldavad soovitud tulemuse listi pea a_1 ja listi sabale vastava samasuguse esituse (vastavalt $a_2 \oplus (\ldots \oplus (a_n \oplus e)\ldots)$ või $a_2 \oplus (a_3 \oplus (\ldots\ldots))))$ kaudu.

Niisugune rekursiooniskeem esineb näiteks muutuja `nullideArv` otsese rekursiooniga definitsioonis (72). Tuletame sellest süstemaatilise analüüsiga argumendivabas stiilis definitsiooni operaatori `foldr` kaudu. Selguse mõttes kirjutame uue definitsiooni nii, et `foldr` argumendid on seotud lokaalsete muutujatega `op` ja `e`.

Lihtne on kirjutada `e` definitsioon. Kuna see argument tuleb `foldr op e` väärtustamise käigus tühja argumentlisti puhul lõppvastuseks, siis võtame definitsiooni paremaks pooleks originaaldefinitsiooni tühja listi juhu parema poole. Antud näites (72) on selleks `0`.

Operaatori `op` definitsiooni kirjutamiseks kujutame ette, et tema argumentideks tulevad operaatori `nullideArv` mittetühja argumentlisti pea, mis originaalkoodis (72) seotakse muutujaga `x`, ja rekursiivse pöördumise tulemus argumentlisti sabal — koodis (72) on see seotud muutujaga `ülejäänu`. Kuna `op` läheb `foldr` argumendiks, siis teda kutsutakse välja ainult sellistes olukordades.

Tulemus peab ses olukorras olema sama mis `nullideArv` rakendamisel kogu argumentlistile — seda aga väljendab originaaldefinitsioonis tingimusavaldis **`if`** `x == 0` **`then`** `1 + ülejäänu` **`else`** `ülejäänu`.

Selle analüüsi tulemuseks on definitsioon

```
nullideArv
  = let
      op x ülejäänu
        = if x == 0 then 1 + ülejäänu else ülejäänu .
      e = 0
    in
    foldr op e
```

(180)

Kõik originaaldefinitsiooniga ühised muutujad on kasutusel samas tähenduses.

Analoogilise töö tulemusel saaksime üldisema operaatori `loenda` definitsioonist (169) koodi

```
loenda p
  = let
      op x ülejäänu
        = if p x then 1 + ülejäänu else ülejäänu .
      e = 0
    in
    foldr op e
```

Argumentoperaatori defineerimisel tuleb hoolas olla, et mitte teha teda oma teise argumendi järgi agaraks, kui see on välditav. Kuna teine argument tuleb rekursiivsest pöördumisest, toimub tema agaruse korral rekursiivne pöördumine juba enne kui argumentoperaatori rakendust üldse väärtustama hakatakse. Rekursiivne pöördumine omakorda aga ei saa anda tulemust enne, kui sealne sama operaatori rakendamine on tulemuse andnud. See nõiaring osutab, et tulemuseni võidakse jõuda alles siis, kui list on lõpuni läbi käidud ja rekursiooni baasjuht töödeldud. Lõpmatu listi puhul aga ei juhtugi seda kunagi ning arvutus jääb midagi välja andmata tsüklisse.

Proovime `foldr` kaudu defineerida muutuja `kaheksLoe`, mille senised definitsioonid on koodides (76), (77), (80), (109). Neist koodis (77) rahuldab rekursiooniskeem tingimust, mis lubab argumendivabale stiilile ülemineku `foldr` abil.

Tühjal listil tuleb (77) järgi välja anda `([] , [])`. Mittetühja argumentlisti pea on koodis (77) seotud muutujaga `x`, rekursiivse pöördumise tulemus argumentlisti

sabal aga näidisega `(us , vs)`. Tulemus ses olukorras on originaaldefinitsiooni järgi `(x : vs , us)`. Seega paistab, et sobib definitsioon

```
kaheksLoe
  = let
      op x (us , vs)
        = (x : vs , us) .
      e = ([] , [])
    in
    foldr op e
```
(181)

Jällegi on kõik originaaldefinitsiooniga ühised muutujad kasutusel samas tähenduses. Kaotades mõneti tarbetud lokaalsed definitsioonid, võime kirjutada ka

```
kaheksLoe
  = foldr (\ x (us , vs) -> (x : vs , us)) ([] , [])
```
(182)

Kuid definitsioonid (181) ja (182), ehkki samaväärsed omavahel, pole samaväärsed originaaldefinitsiooniga (77). Et definitsioonides (181) ja (182) sattus `foldr` argumentoperaatori definitsiooni teise argumendi kohale agar näidis `(us , vs)`, jääb arvutus nende definitsioonide järgi lõpmatu argumentlisti korral lõpmatusse tsüklisse ilma mingigi väljundita. Definitsiooniga (77) arvutades tekib tulemusena alati paar kahest listist.

Olukorra parandamiseks piisab probleemne paarinäidis laisaks muuta, et operatsiooni täitmisega alustataks enne rekursiivset pöördumist. Saame definitsiooni

```
kaheksLoe
  = let
      op x ~(us , vs)
        = (x : vs , us)
      e = ([] , [])
    in
    foldr op e
```

või, samaväärselt ilma lokaalsete deklaratsioonideta,

```
kaheksLoe
  = foldr (\ x ~(us , vs) -> (x : vs , us)) ([] , [])
```

See ongi laisa näidise tüüpilisim praktiline kasutus.

Esialgne mittevastavus originaaldefinitsiooniga (77) tekkis sellest, et seal oli rekursiivse pöördumise tulemus seotud lokaalse deklaratsiooniga ja lokaalsete deklaratsioonide poolt seotavaid näidiseid käsitletakse alati laisalt. Seda on eespool ka selgitatud: avaldiste (42) ja (43) väärtustamine käib erinevalt.

Vaatame veel koodiga (148) või (155) defineeritud operaatorit `osalistid`. Ka need võib `foldr` abil argumendivabas stiilis ümber teha; võtame siin aluseks (155).

Tulemus tühjal listil on `[[]]`, mittetühja listi pea ja rekursiivse pöördumise tulemus sabal seotakse seal vastavalt muutujatega `x` ja `pool`. Kirjutame argumendivabas stiilis ümber ka lambdaavaldise `\ as ass -> (x : as) : ass`, kust saame `(:) . (x :)`. Tulemuseks on definitsioon

```
osalistid
  = foldr (\ x pool -> foldr ((:) . (x :)) pool pool) .
    [[]]
```

Operaatori `foldl` rakendamine abstraheerib selliseid ühe akumulaatoriga arvutusi listil, kus akumulaatori uus väärtus leitakse vanast, rakendades talle ja listi jooksvale elemendile mingit kindlat operatsiooni, ja lõpus antakse akumulaator välja. See ilmneb esitusest (151): andmele e (vasteks akumulaatori algväärtus) rakendub opereerimine listi peaga, järgneb aga analoogne protsess, kus algandme rollis on selle operatsiooni tulemus $e \oplus a_1$ (vasteks akumulaatori uus väärtus) ja listi rollis algse listi saba.

Muutuja `nullideArv` definitsioon (98) annab just seda tüüpi arvutuse. Tuletame sellest süstemaatilise mõttekäiguga argumendivabas stiilis definitsiooni operaatori `foldl` kaudu. Jällegi seome `foldl` argumendid selguse mõttes lokaalsete muutujatega `op` ja `e`.

Muutuja `e` defineerimisel arvestame, et see on akumulaatori algväärtus. Koodis (98) tuleb loendur-akumulaatori `n` algväärtus lokaalse operaatori `arv` algsest väljakutsest, mis annab tema kohale `0`.

Muutuja `op` defineerimiseks kujutame ette, et tema argumentideks on akumulaator ja mittetühja argumentlisti pea. Koodis (98) märgivad neid vastavalt muutujad `n` ja `z` lokaalse operaatori definitsioonis. Tulemus peab olema akumulaatori värskendatud väärtus ehk originaalkoodi mõttes `n` väärtus järgmisel rekursiivsel väljakutsel, mis on antud tingimusavaldisega `if z == 0 then n + 1 else n`.

See analüüs annab tulemuseks koodi

```
nullideArv
  = let
      op n z
        = if z == 0 then n + 1 else n ,
      e = 0
    in
    foldl op e
```

mis erineb operaatoriga `foldr` kirjutatud koodist (180) märksa vähem kui originaaldefinitsioonid (98) ja (72) üksteisest. Erinevused on muutujate nimedes (`x`

asemel `z` ja `ülejäänu` asemel `n`) ja plussi argumentide järjekorras, mis pole põhimõttelised. Lisaks on `op` argumentide järjekord vastupidine. See tuleneb operaatorite `foldr` ja `foldl` tüüpide erinevusest, mis on keele loojate suvaotsus.

Ka varem antud muutuja `suurSumma` definitsioonide (154) ja (105) samaväärsus ei piirdnud ainult samade tulemustega samadel argumentidel, vaid ka arvutusskeem oli sarnane, akumuleeriv. Isegi muutujad `a` ja `i` on neis koodides sama tähendusega.

Kui akumulaatorit teisendatakse ka rekursiooni baasjuhul, siis on operaatorit `foldl` kasutav argumendivabas stiilis definitsioon võimalik, kui lisada kompositsioon operaatori või (argumendivaba) avaldisega, mis seda teisendust väljendab. Kui akumulaatoreid on mitu või on akumulaatorite kõrval veel abiparameetreid, siis tuleb nad ühte andmestruktuuri kokku pakkida.

Kirjutame näiteks koodiga (112) antud muutujale `nullsummadeArv` argumendivabas stiilis definitsiooni. Selguse mõttes defineerime `foldl` argumendid jälle lokaalsete muutujate `op` ja `e` väärtuseks. Kuna originaalkoodi abifunktsioonis on kaks lisaparameetrit `s` ja `n`, siis kasutame paariakumulaatorit, mille komponendid vastavad neile parameetritele.

Muutujate `s` ja `n` algväärtused koodis (112) tulevad lokaalse operaatori algsest väljakutsest, mis annab mõlema parameetri kohale `0`. Mittetühja argumentlisti pea seotakse muutujaga `z`. Argumentide `s` ja `n` kohale järgneval rekursiivsel väljakutsel lähevad vastavalt `s'` ja **`if`** `s' == 0` **`then`** `n + 1` **`else`** `n`, kus `s'` on defineeritud omakorda lokaalse deklaratsiooniga, mille võib lihtsalt üle võtta.

Kuna originaalkoodi abifunktsioon annab lõpus välja vaid parameetri `n` väärtuse, tuleb samaväärse käitumise tekitamiseks nüüd lõpus akumulaatorist teine komponent võtta. Kogu analüüsi tulemusena tekib kood

```
nullsummadeArv
  = let
      op (s , n) z
        = let
            s'
              = s + z                          .
          in
          (s' , if s' == 0 then n + 1 else n)
      e = (0 , 0)
    in
    snd . foldl op e
```

Ülesandeid

384. Otsida Hugsi teegist üles muutujate `concat`, `and`, `or`, `reverse` definit-

sioonid ja saada neist aru.

385. Millise tuntud operaatoriga on samaväärne avaldis `flip (foldr (:))`?

386. Kirjutada `foldr` abil argumendivabas stiilis definitsioon koodiga (84) defineeritud muutujale `kordaTähed`.

387. Lahendada uuesti ülesanded 188, 189, 191, 192, 203, 314, kirjutades definitsioonid argumendivabas stiilis `foldr` abil.

388. Operaatori `foldr` abil argumendivabale stiilile üle minnes defineerida muutuja `esimMittenull` väärtuseks funktsioon, mis võtab argumendiks arvude listi: kui selles leidub nullist erinevaid elemente, siis annab välja esimese neist, vastasel korral kui list on lõplik, siis annab väärtuseks 0.

389. Operaatori `foldr` abil argumendivabale stiilile üle minnes defineerida muutuja `loeFilter` väärtuseks funktsioon, mis võtab järjest argumendiks predikaadi p ja listi l ning annab välja paari, mille esimene komponent on list neist l elementidest, mis rahuldavad predikaati p, ja teine komponent on arv, kui palju on selles listis vähem elemente kui listis l.

390. Vaatleme definitsiooni

```
dropWhileLõpust p
  = reverse . dropWhile p . reverse
```

Anda muutujale `dropWhileLõpust` uus definitsioon argumendivabas stiilis operaatori `foldr` abil, nii et iga lõpliku argumentlisti korral töötab uus versioon samamoodi kui varasem. Lisaks peab uus versioon töötama teatud olukorras ka lõpmatu listi jaoks.

391. Kirjutada definitsiooniga (78) antud muutujale `eraldaSegment` samaväärne argumendivabas stiilis definitsioon operaatori `foldr` abil.

392. Defineerida kahel viisil argumendivabas stiilis muutuja `sgnSuhe` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab tulemuseks GT, LT või EQ vastavalt sellele, kas positiivseid elemente on rohkem kui negatiivseid, negatiivseid on rohkem kui positiivseid või on neid võrdselt.

393. Defineerida muutuja `loenda` argumendivabas stiilis operaatori `foldl` abil, nii et arvutus käiks samamoodi kui definitsiooni (170) korral.

394. Argumendivabas stiilis defineerida muutuja `keera` väärtuseks funktsioon, mis võtab argumendiks listide listi ja, interpreteerides seda tabelina (algse listi elemendid on tema read), annab tulemuseks 90° päripäeva pööratud tabeli

samal kujul. Võib eeldada, et tabel on lõplik, sisaldab vähemalt ühe rea ning kõik read on ühepikkused.

395. Operaatori `foldl` abil argumendivabale stiilile üle minnes defineerida muutuja `osasummadSellisedEt` väärtuseks funktsioon, mis võtab argumendiks arvutüübil töötava predikaadi p ja arvude listi l ja annab tulemuseks listi, mille elementideks on väiksemast suuremani need positiivsed arvud n, mille korral listi l esimese n elemendi summa rahuldab predikaati p. Lõpmatul listil ei pea töötama.

396. Vaatame definitsiooni

```
concatLõpust
  = reverse . concat˙
```

Anda muutujale `concatLõpust` uus samaväärne võimalikult lühike definitsioon, mille korral arvutus vahestruktuure ei moodusta.

397. Operaatori `foldl` abil argumendivabale stiilile üle minnes defineerida muutuja `viimaseNullsummani` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab tulemuseks selle algusosa kuni viimase sellise elemendini x, mille korral elementide summa algusest kuni elemendini x on 0 (kui sellist elementi x ei leidu, siis annab tulemuseks tühja listi).

5.3 Veel abstraheerimisest

5.3.1 Süntaktiline abstraheerimine

Tihti saab olemasolevast koodist abstraktsema lihtsalt mehhaanilise teisendamisega. Nii on lugu näiteks juhul, kui olemasolevas koodijupis (avaldises või definitsioonis) on teada muutuja, mille kohal tahetakse näha vabalt muutuvat parameetrit.

Senisteski näidetes on abstraktsema koodi kirjutamiseks kasutatud eeskujuna mõnd olemasolevat koodijuppi. Näiteks definitsioon (169) saadi definitsiooni (72) ja definitsioon (170) definitsiooni (98) ümbertöötamisel. Need muutused polnud siiski üleni mehhaanilised.

1. Abstraheerimine muutujast avaldises. Igast avaldisest on võimalik ükskõik milline defineeritud muutuja argumendina välja eraldada. Täpsemalt, iga avaldise $\mathfrak{e}$ saab põhimõtteliselt teisendada samaväärseks avaldiseks kujul $\mathfrak{e}'\ \mathfrak{x}$, kus $\mathfrak{x}$ on nõutud muutuja ja $\mathfrak{e}'$ on avaldis, mis muutujat $\mathfrak{x}$

samas tähenduses ei sisalda. Seda protsessi nimetatakse avaldise $\mathfrak{e}$ **abstraheerimiseks** muutujast $\mathfrak{x}$.

Avaldis $\mathfrak{e}'$ on siis abstraktsem kui $\mathfrak{e}$. Tema abil saab kirjeldada $\mathfrak{e}$ väärtuse, andes argumendiks $\mathfrak{x}$, kuid argumenti varieerides võib kirjeldada soovi korral palju muud.

Puhtmehhaaniline viis $\mathfrak{e}'$ saamiseks on võtta $\mathfrak{e}'$ = `\ x -> e`. Muutuja $\mathfrak{x}$ omandab lambda all avaldises $\mathfrak{e}$ uue tähenduse formaalse parameetrina. Rakendamine `(\ x -> e) x` on samaväärne avaldisega $\mathfrak{e}$, sest rakendamise väärtus on $\mathfrak{e}$ väärtus eeldusel, et lokaalse $\mathfrak{x}$ väärtus võrdub välise $\mathfrak{x}$ väärtusega, millega ta avaldises $\mathfrak{e}$ algselt oligi.

Selguse mõttes võib lambdaavaldises kõik $\mathfrak{x}$-d mõne uue muutuja vastu vahetada, sisu sellest ei muutu. Samuti võiksime lambdaavaldise asemel defineerida mõne uue muutuja, mille väärtus oleks sama funktsioon mis sel lambdaavaldisel, ja kasutada seda.

Näiteks kui $\mathfrak{e}$ = `0 : zs` ja tahame abstraheerida muutujast `zs`, siis võime mehhaaniliselt võtta $\mathfrak{e}'$ = `\ zs -> 0 : zs`. Samaväärne oleks võtta näiteks $\mathfrak{e}'$ = `\ xs -> 0 : xs` ning muidugi sobivad ka loomulikumad valikud nagu näiteks sektsioon $\mathfrak{e}'$ = `(0 :)`.

Kui avaldis $\mathfrak{e}$ juba on õigele muutujale rakendamise kujul, siis midagi erilist vaja teha ei ole: näiteks avaldist `log x` muutuja `x` järgi abstraheerides võib kirjutada lihtsalt `log`, ehkki muidugi sobib ka mehhaaniline `\ x -> log x`. Kuid pangem tähele, et abstraheerida saab ka muutujast `log`, mis annab lambdaavaldise `\ log -> log x`. Selle väärtus on funktsioon, mis võtab argumendiks suvalise (tüübilt sobiva) funktsiooni ja annab tulemuseks tema väärtuse argumendil, mis võrdub muutuja `x` väärtusega. Kui argumendiks anda logaritmfunktsioon, saame logaritmi `x` väärtusest, st täpselt endise avaldise `log x` väärtuse.

Kogu põhimõte on sama nagu standardses olukorras, kus avaldise `x * x` kasutamise asemel tuuakse koodiga (21) või (37) sisse operaator `sqr` ja kirjutatakse `sqr x`.

Teisendus toimib sõltumata sellest, mitmes kohas muutuja $\mathfrak{x}$ avaldises $\mathfrak{e}$ esineb. Ta ei pea seal üldse esinema; sellisel juhul tekib konstantne funktsioon.

Võiksime avaldist `log x` abstraheerida ka muutujast `zs`; saaksime lambdaavaldise `\ zs -> log x`, mis on samaväärne avaldisega `const (log x)`.

On ilmne, et kui muutujal $\mathfrak{x}$ on väärtus tüüpi X ning avaldise $\mathfrak{e}$ väärtus on sel juhul tüüpi Y, siis abstraheeriva avaldise väärtus on $X \to Y$.

Näiteks kui `x` väärtus on tüüpi Double, siis `log x` väärtus on samuti tüüpi Double ning abstraktsioon seega tüüpi Double $\to$ Double.

Kui aga logaritmist abstraheerida, siis et selle tüüp on Double $\to$ Double, siis abstraktsiooni tüüp on (Double $\to$ Double) $\to$ Double.

Kui algne avaldis $\mathfrak{e}$ pole funktsioonitüüpi, siis avaldise $\mathfrak{e}'$ väärtuseks olev funktsioon ei ole karritatud, vastasel juhul aga on karritatud. Üldiselt $\mathfrak{e}'$ võtab täisargumenteeritult täpselt ühe argumendi rohkem kui $\mathfrak{e}$.

Abstraheerides avaldist `\ x y -> a + b / 2` jagamisoperaatorist, võime tulemust väljendada ühe võrra suurema argumendinäidiste arvuga lambdaavaldisega `\ (/) a b -> a + b / 2`.

Ülesandeid

398. Kirjutada avaldis, mis abstraheerib avaldist `filter isLower "Oooo!"` muutujast `isLower`. Defineerida võimalikult tavalisel viisil uus muutuja, mille väärtus oleks sama funktsioon.

2. Abstraheerimine muutujast definitsioonis. Kui muutuja, millest soovitakse abstraheerida, asub definitsiooni paremas pooles, on sama tehnikaga tavaliselt võimalik vahetult tuletada abstraktsem definitsioon. Kui definitsioon pole rekursiivne, siis piisab muutuja lisamisest vasakusse poolde formaalsete parameetrite hulka.

Rekursiivse definitsiooni korral tuleb lisaks hoolitseda, et argumentide arv vasakul ja rekursiivses pöördumises jääks klappima. Selleks tuleb muutuja, millest abstraheeritakse, lisada ka rekursiivsetesse pöördumistesse.

Abstraheerime näiteks liitmisoperaatorist `+` muutuja `kahesummad` definitsioonis (75). Olgu uus ja abstraktsema operaator `kaheOp`. Lisaks muutujanime vahetusele piisab mõlema deklaratsiooni vasakusse poolde ning rekursiivsesse pöördumisse lisada argumendina `(+)`; et teise deklaratsiooni parem pool operaatorit `+` ei sisalda, siis võib seal vasakus pooles kasutada jokkerit. Nii saame definitsiooni

```
kaheOp (+) (x : xs@ (y : _))
  = x + y : kaheOp (+) xs
kaheOp _   _                  ,
  = []
```
(183)

kus `+` tähistab juba muutuvat argumentoperatsiooni, mitte liitmist. (Selle oleks võinud asendada vabalt valitud uue muutujaga, kuid praegune variant võimaldab vana koodi maksimaalselt ära kasutada.)

Argumentide hulka lisandunud kahe argumendiga operaatori argumendid tulevad samast listist, seega nad peavad olema sama tüüpi. Kuid kood (183) ei nõua kuskil, et tulemuslist ja argumentlist oleksid sama tüüpi elementidega; seetõttu saab argumentoperaatori väärtusetüüp olla midagi muud. Näiteks avaldise `kaheOp (<=)` väärtus on funktsioon, mis argumentlisti järgi annab välja niisuguse tõeväärtuste listi, kus True esineb sellistes kohtades, kus originaallistis on kaks kõrvutiasuvat elementi järjestatud mittekahanevalt, ja False muudes kohtades.

Muutuja `kaheOp` signatuuriks saame

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

Üleminek on analoogiline näiteks sellele, kuidas muutuja `nullideArv` definitsioonist (72) sai muutuja `loenda` definitsioon (169). Kuid seal ei abstraheeritud olemasolevast muutujast ega konstruktorist, vaid mõttelisest predikaadist, mis kontrollib argumendi võrdumist nulliga.

Et saada analoogia täielikuks, võiks vahesammuna tuua sisse muutuja `p` koodiga

```
p :: (Num a)
  => a -> Bool'

p = (0 ==)
```

ning kirjutada definitsioon (72) muutuja `p` kaudu ümber kujule

```
nullideArv (x : xs)
  = let
      ülejäänu
        = nullideArv xs
    in
    if p x then 1 + ülejäänu else ülejäänu
nullideArv _
  = 0
```

Siit on definitsioon (169) juba mehhaaniliselt tuletuv.

Ülesandeid

399. Kirjutada definitsioon, mis abstraheerib operaatori `põimiTase` definitsioonis (127) muutujast `põim`. Olgu uus operaator `kahekaupaMap`.

400. Kirjutada ümber muutuja `põimiPuu` definitsioon (128), kasutades ülesandes 399 defineeritud operaatorit `kahekaupaMap`. Abstraheerida saadud definitsioon muutujast `põim`.

     Uue operaatori rakendamise kaudu realiseerida uuesti järjestamine põimemeetodil. Katsetada uut operaatorit ka teistsuguste argumentidega.

5.3.2 Abstraheerimise kaugemad eesmärgid

Tihti saab esmapilgul lausa täiesti erinevad ülesanded lahendada ühe ja sama polümorfse operaatori rakendusena, kus operaatori väärtus on kõrgemat järku funktsioon. Lahenduste põhiosa realiseerib siis üks ja sama kood — selle operaatori definitsioon. Meisterlikkus funktsionaalses programmeerimises eeldab selliste ühiste arvutusskeemide läbinägemist ja väljatoomist.

Ühe ja sama koodi lai kasutatavus teenib vähemalt kaht tähtsat eesmärki.

1. Programmeerimise lihtsustamine. Kui paljude ülesannete lahenduste ühine tuum on realiseeritud, siis nõuab üksikülesannete lahendamine juba palju vähem tööd. Võib piisata kõrgemat järku funktsiooni olemasolu teadvustamisest, et märgata tema rakendusi, mille peale muidu poleks tulnud.

Koodiga (130) defineerisime muutuja `aste` väärtuseks arvude astendamise täisarvuga. See kasutas asjaolu, et astendamine esitub ühe ja sama arvu paljukordse korrutamise kaudu.

Fikseeritud objektiga sama operatsiooni paljukordse sooritamise näiteid on matemaatikas palju — üks silmatorkavaimatest korrutamine, mis esitub samal moel liitmise kaudu —, mistõttu on mõttekas see algoritmiskelett välja abstraheerida.

Defineerime muutuja `kõrgAste`, millega saaks arvutada mistahes etteantud üht tüüpi argumentidega binaarse operatsiooni naturaalarv korda sooritamise tulemusi. Tema väärtuseks peaks saama kõrgemat järku funktsioon, mis enne astme alust ja astendajat võtab argumendiks binaarse operatsiooni, millega kokkuarvutamine toimub, ja erilise andme, mis tuleb vastuseks astendaja 0 puhul. Jätame vaatluse alt välja astendamise negatiivse täisarvuga, mille tarvis oleks vaja veel ka pöördoperatsiooni. Sobib signatuur

```
kõrgAste
  :: (Integral b)                          .
  => (a -> a -> a) -> a -> a -> b -> a
```

Põhiosas koodi (130) abstraheerimisega muutujast `*` ja konstruktorist `1` saame

```
kõrgAste (*) e a n
  = case compare n 0 of
      GT
        -> a * kõrgAste (*) e a (n - 1)
      EQ                                          ,   (184)
        -> e
      _
        -> error "kõrgAste: neg. astendaja"
```

kus `kõrgAste` esimene argument `(*)` tähistab operatsiooni ja `e` arvuga 0 astendamise tulemust. Argumendid `a` ja `n` on, nagu ka koodis (130), astme alus ja astendaja.

Tavalise täisarvulisse astmesse tõstmise operaatori saab nüüd defineerida erijuhuna kõrgemat järku astendamisest koodiga

```
aste
  = kõrgAste (*) 1'
```
(185)

millele seoses negatiivse astendaja puudumisega võib anda originaalversiooniga võrreldes üldisema signatuuri

```
aste
  :: (Num a, Integral b).
  => a -> b -> a
```

Kood (185) fikseerib kõrgemat järku astendamisel korratavaks operatsiooniks tõelise korrutamise ja 0-ga astendamise tulemuseks arvu 1.

Soovi korral võime defineerida teisi analoogilisi operaatoreid, andes `kõrgAste` argumentideks midagi muud. Näiteks saab täisarvuga korrutamise defineerida korduva liitmisoperatsiooni kaudu koodiga

```
korrutis
  = kõrgAste (+) 0
```
(186)

ja superastendamise korduva tavaastendamise kaudu koodiga

```
super
  = kõrgAste (**) 1'
```
(187)

kus signatuurideks sobivad

```
korrutis
  :: (Num a, Integral b),
  => a -> b -> a

super
  :: (Floating a, Integral b).
  => a -> b -> a
```

Kui `kõrgAste` on kutsutud välja definitsioonist (186), siis definitsiooni (184) lokaalse muutuja `*` väärtus on liitmine, kui aga definitsioonist (187), siis ujukomaarvude astendamine.

Koodi (184) korrektseks kasutamiseks on vaja `e` väärtuseks valida `*` väärtuseks oleva operatsiooni parempoolne ühik, st objekt, millega operatsiooni sooritamine paremalt ei avalda mõju. Näiteks väljakutsel koodist (185) saab selleks korrutamise

ühik 1, väljakutsel koodist (186) aga liitmise ühik 0. Superastendamise definitsiooni (187) puhul on selleks jälle 1; arv 1 on astendamise parempoolne ühik, sest arv astmes 1 on alati arv ise.

Kui seda printsiipi mitte järgida, hakkab `e` väärtus tulemust mõjutama, sest viimasel rekursioonisammul sooritatakse operatsioon temaga.

Tuntud on matemaatikas ka funktsioonide astendamine, kus korratavaks operatsiooniks on kompositsioon. Kompositsiooni ühik on samasusfunktsioon, mis tuleb moodulist Prelude muutujas `id`. Seega võime kirjutada näiteks avaldise `kõrgAste (.) id tail 2`, mille väärtus on listi saba võtmise funktsiooni teine aste ehk kompositsioon iseendaga. Teisi sõnu on see saba võtmise funktsiooni kahekordne rakendamine ehk saba saba võtmine. Et tegu on funktsiooniga, saame tõrgeteta lisada argumendi, näiteks avaldise

```
kõrgAste (.) id tail 2 [1, 2, 3, 4, 5]
```

väärtus on lõplik list elementidega 3, 4, 5.

Arvude astendamisest olid meil olemas ka efektiivsemad variandid (131) ja (132), mis kasutasid "jaga ja valitse" strateegiat koos dünaamilise programmeerimisega. Ka need on abstraheeritavad samamoodi nagu kood (130). Lähtudes koodist (131), saame muutujale `kõrgAste` uue definitsiooni

```
kõrgAste (*) e a n
  = case compare n 0 of
      GT
        -> let
             (q , r)
               = n `divMod` 2
             x = kõrgAste (*) e a q
           in
           if r == 0 then x * x else a * x * x
      EQ
        -> e
      _
        -> error "kõrgAste: neg. astendaja"
```
. (188)

Definitsiooni (188) korral töötavad koodiga (185) defineeritud operaator `aste` ja koodiga (186) defineeritud operaator `korrutis` palju kiiremini kui endise definitsiooni (184) korral. Operatsioonide arv on võrdeline astendaja logaritmiga; samas mida suurem on astendaja, seda suuremaks lähevad arvutuse käigus argumendid ja seda aeganõudvam on üldjuhul operatsiooni ühekordne sooritamine.

Eriti oluline on arvestada, et "jaga ja valitse" strateegiaga on astet võimalik arvutada vaid tänu korrutamisoperatsiooni assotsiatiivsusele, sest selline astendamisalgoritm

baseerub sulgude ümberpaigutamisel. Seega annab definitsiooniga (188) arvutamine korrektseid tulemusi vaid assotsiatiivsete argumentoperatsioonide korral.

Näiteks ei ole võimalik koodiga (188) defineeritud funktsiooni `kõrgAste` rakendusena saada superastendamist, sest astendamine ei ole assotsiatiivne. Kood (187) töötaks lihtsalt valesti.

Seevastu funktsioonide kompositsioon on assotsiatiivne, seega funktsioonide astmete arvutamine töötab korrektselt ka koodiga (188). Kuid funktsioonide astendamise juures ei anna astendamiseks "jaga ja valitse" strateegia kasutamine reaalselt mingit võitu efektiivsuses. Kompositsioonide arv tuleb küll astendaja suhtes logaritmiline, kuid antud juhul pole see oluline näitaja, sest kasutajat ei huvita mitte funktsiooni aste ise, vaid tema rakendamise tulemus teatavale argumendile.

Kui funktsiooni aste on esitatud kujul $g \circ g$, siis erinevalt tavalise astendamise juhust pole siin mingit kasu asjaolust, et operatsiooni argumendid on võrdsed, sest seda kompositsiooni mingile argumendile rakendades sooritatakse kõigepealt üks g sellel argumendil ja teine g saadud tulemusel, mitte samal argumendil, mis võimaldaks arvutusressurssi kokku hoida. Kokkuvõttes sooritatakse mingi funktsiooni f mingi astme rakendamisel argumendile ikka astendajaga võrdne arv f rakendamisi.

Pöördume nüüd veel järjekordselt tagasi Fibonacci jada liikmete arvutamise juurde. Kas kõrgemat järku astendamine võiks ka siin aidata?

Osutub, et huupi püstitatud küsimuse vastus on jaatav. Võtmerolli mängib tähelepanek, et kõik Fibonacci arvud on võimalik kätte saada maatriksi $\mathscr{F}$ astmetest, kus

$$\mathscr{F} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}.$$

Täpsemalt, kehtib seaduspära

$$\mathscr{F}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}. \tag{189}$$

Tõepoolest, see ilmselt kehtib $n = 1$ korral ning kui ta kehtib $n = i$ jaoks, siis

$$\begin{aligned} \mathscr{F}^{i+1} &= \mathscr{F} \cdot \mathscr{F}^i \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i-1} + F_i & F_i + F_{i+1} \end{pmatrix} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i+1} & F_{i+2} \end{pmatrix}, \end{aligned}$$

st ta kehtib ka $n = i + 1$ jaoks.

Seega tuleks realiseerida 2×2-maatriksite korrutamine. Selle töö lihtsustamiseks tasub teha veel üks tähelepanek. Kirjutades võrduses (189) F_{n-1} kohale samaväärselt $F_{n+1} - F_n$, me näeme, et $\mathscr{F}$ kõik astmed on kujul

$$\begin{pmatrix} y - x & x \\ x & y \end{pmatrix}, \tag{190}$$

kus x, y on mingid täisarvud. Järelikult on meisse puutuvate maatriksite ülemine rida alumise reaga üheselt määratud ja kõik vajalikud tegevused saab kirjeldada maatriksite alumiste ridade peal.

Maatriksist $\mathscr{F}$ jääb järele $\begin{pmatrix}1 & 1\end{pmatrix}$. Arvuga 0 astendamine annab ühikmaatriksi, mille alumine rida on $\begin{pmatrix}0 & 1\end{pmatrix}$. Lõpuks, kui meil on kaks maatriksit kujul (190), mille alumised read on vastavalt $\begin{pmatrix}a & b\end{pmatrix}$ ja $\begin{pmatrix}c & d\end{pmatrix}$, siis korrutismaatriksi alumise veeru vasak komponent tuleb $a(d-c)+bc$ ehk $ad+bc-ac$ ning parem komponent tuleb $ac+bd$.

Vastavalt valemile (189) on Fibonacci jada liige järjekorranumbriga n leitav maatriksi $\mathscr{F}^n$ alumise rea vasaku komponendina. Esitades kahekomponendilisi ridu paarina, saame kõige selle põhjal otsitavaks definitsiooniks

```
fib n
  | n >= 0
    = let
        (a , b) **+* (c , d)
          = (a * d + b * c - a * c , a * c + b * d) .
      in
      fst (kõrgAste (**+*) (0 , 1) (1 , 1) n)
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```

(191)

Kuna maatriksite korrutamine on assotsiatiivne ja meie operatsioon paaridel peegeldab teatavate maatriksite korrutamist üksühele, on ka see operatsioon paaridel assotsiatiivne ja on võimalik kasutada `kõrgAste` definitsiooni (188). Siis on Fibonacci arvu leidmiseks koodiga (191) tehtav operatsioonide arv laias laastus võrdeline järjekorranumbri logaritmiga.

See on kolossaalne hüpe kiiruses. Meenutagem, et muutuja `fib` varasematest definitsioonidest esimene, naiivsel matemaatilise seose ümberkirjutamisel põhinev (67), andis eksponentsiaalse töömahu, kõik ülejäänud ((68), (106) ja (115)) aga lineaarse. Kood (191) on parem isegi koodist (93), mis lasi Fibonacci arve valmisarvutatud listist otsida, sest ka listist otsimiseks tehtava töö maht kasvab lineaarselt.

Arvutuskiiruse kasvus ei ole abstraheerimisel muidugi otseselt mingit osa — sama algoritmi võinuks realiseerida ka otse rohujuure tasandil. Kuid tõsiasi on, et mõeldes astendamisest kõrgemal abstraktsioonitasemel, on selle lahenduse idee peale tulek lihtsam, samuti olles astendamisalgoritmi üldisel tasemel realiseerinud, nõuab selliste lahenduste programmeerimine vähem tööd.

On huvitav märkida, et kui kasutada siiski `kõrgAste` lineaarse töömahuga definitsiooni (184), siis arvutuse käigus läbitakse täpselt samad vahetulemused nagu näiteks definitsiooni (68) korral, mis itereeris operatsiooni, mis leiab paari (a, b) järgi järgmise paari $(b, a + b)$. Tõepoolest, kuna astme alus on $(1\,, 1)$, siis paarist

(a, b) järgmine aste on parajasti $(1b + 1a - 1a, 1a + 1b)$ ehk $(b, a + b)$.

Viimaks vaatame ülesannet defineerida muutuja `bitisõned` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see pole negatiivne, siis annab tulemuseks listi, mille elemendid on parajasti kõik numbritest 0 ja 1 koosnevad sõned pikkusega n; signatuuriks sobib

```
bitisõned
  :: (Integral a) .
  => a -> [String]
```

Avastame ka selles puhtalt kombinatoorses ülesandes astendamise arvutusskeemi ja kirjutame lahenduse kõrgemat järku astendamise kaudu.

Paneme tähele, et kui meil on õnnestunud leida list, mille elementideks on kõik bitisõned pikkusega n, siis selleks, et leida kõigi ühe võrra pikemate bitisõnede listi, tuleb lisada kõigile bitisõnedele pikkusega n ette kord 0, kord 1, ja moodustada tulemustest üks list.

Et vaadelda seda tegevust teatava binaarse operatsiooni rakendamisena, mille argumendid oleksid üht tüüpi, moodustame ka üksikutest bittidest bitisõnede listi k elementidega "0", "1". Vastavalt äsja kirjeldatud meetodile kindla pikkusega bitisõnede listist l ühe võrra pikemate bitisõnede saamiseks peaks operatsioon, kui argumendiks on listid k ja l, valima kõikvõimalikel viisidel kummastki ühe elemendi (sõne) ja omavahel konkateneerima ning andma välja listi nii saadud sõnedest.

Binaarne operatsioon sõnede listidel, milleni jõudsime, on sarnane hulkade otsekorrutamisega; vahe on esiteks selles, et hulkade rollis on listid, ja teiseks selles, et erinevaist kogumitest võetud sõned paaripaneku asemel konkateneeritakse.

Kuna konkatenatsiooni rollis võib kergesti ette kujutada muid operatsioone, siis defineerime kõigepealt operaatori `otsekorruta` väärtuseks kõrgemat järku funktsiooni, mis võtab argumentideks binaarse operatsiooni $\oplus$ ja kaks listi ning annab tulemuseks "otsekorrutise", kus kummastki argumentlistist kõikvõimalikel viisidel võetud elemendid opereeritakse omavahel tehtega $\oplus$. Signatuur ja definitsioon on

```
otsekorruta
  :: (a -> b -> c) -> [a] -> [b] -> [c]'

otsekorruta (*) xs ys
  = [x * y | x <- xs, y <- ys]'
```
(192)

Nüüd näiteks `otsekorruta (,) [1, 2, 3] "AB"` annab listi paaridest $(1, \mathrm{A})$, $(1, \mathrm{B})$, $(2, \mathrm{A})$, $(2, \mathrm{B})$, $(3, \mathrm{A})$, $(3, \mathrm{B})$. Meil aga on vaja sõnesid paaripaneku asemel konkateneerida, mistõttu läheb käiku rakendus `otsekorruta (++)`. Muutuja `bitisõned` otsitav definitsioon on

```
bitisõned
  = kõrgAste (otsekorruta (++)) [""] ["0", "1"]'
```
(193)

Pole raske veenduda, et assotsiatiivsele operatsioonile vastav otsekorrutamine on samuti assotsiatiivne. Et konkatenatsioon on assotsiatiivne, siis järelikult koodi (193) poolt välja kutsutav operaator `kõrgAste` võib olla defineeritud koodiga (188). Siiski, sarnaselt funktsioonide astendamisega, ei anna see siin võitu efektiivsuses.

Ülesandeid

401. Kasutades koodiga (184) või (188) defineeritud muutujat `kõrgAste`, kirjutada interaktiivse interpretaatori käsurealt avaldis, mille väärtus saadakse siinuse 100-kordsel rakendamisel arvule 1.

402. Kirjutada uus lahendus ülesandele 287, kasutades koodiga (188) antud muutujat `kõrgAste`.

403. Anda koodiga (71) defineeritud muutujale `sq2` samaväärne definitsioon, mille järgi arvutades oleks operatsioonide arv argumendi suhtes logaritmiline.

404. Kasutades koodiga (188) antud muutujat `kõrgAste`, defineerida muutuja `posArvud` väärtuseks funktsioon, mis võtab argumendiks numbrite listi ja täisarvu n ning annab tulemuseks listi järjekorras kõigist n numbriga kirjutatud arvudest positsioonilisel kujul, kus numbrid tulevad antud listist (arv võib alata ka nulliga). Võib eeldada, et numbrite listis ei ole kordumisi ja $n \geqslant 0$.

    Anda koodiga (193) defineeritud muutujale `bitisõned` võimalikult lühike uus definitsioon muutuja `posArvud` kaudu.

405. Defineerida muutuja `kõrgFac` väärtuseks kõrgemat järku funktsioon, mis võtab järjest argumendiks operatsiooni, tema parempoolse ühiku ja naturaalarvu n ning arvutab arvud $n, n-1, \ldots, 1$ selle operatsiooniga kokku. Näiteks avaldise `kõrgFac (^) 1 n` väärtus, kui `n` väärtus on naturaalrv n, peab olema $n^{(n-1)^{\cdots^1}}$.

    Operaatori `kõrgFac` kaudu defineerida uuesti samaväärselt ülesandes 182 defineeritud muutuja `fac` ning anda muutuja `kolmnurk` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab välja n-nda kolmnurkarvu (st ülesandes 247 defineeritud listi liikme nr n; negatiivse argumendi korral võib töö lõppeda veateatega).

406. Defineerida muutuja `tabel` väärtuseks funktsioon, mis võtab argumendiks binaarse operatsiooni $\oplus$ ja koostab lõpmatu $\oplus$-tabeli naturaalarvudel. Listi nr m element nr n peab olema $m \oplus n$.

    Kasutades seda muutujat ja lisaks ülesandes 328 defineeritud muutujat `diagPööre`, defineerida muutuja `paarid` väärtuseks list, mille elementideks on kõik naturaalarvupaarid.

Kasutades lisaks juba mainitud operaatorit `diagPööre` ja ülesandes 314 defineeritud operaatorit `elimKorduvad`, defineerida muutuja `posRats` väärtuseks list, milles esineb iga positiivne ratsionaalarv täpselt ühe korra.

407. Anda muutuja `lõiguPoolitamine` väärtuseks kõrgemat järku funktsioon, mis võtab argumendiks funktsiooni ja leiab tema nullkoha lõigu poolitamise meetodiga. Kood peab töötama juhul, kui argumentfunktsioon on kasvav ja tal on nullkoht.

    Kasutades seda muutujat, lahendada uuesti ülesanne 275.

2. Korrektsuse kindlustamine. Testides operaatoreid, mille väärtus on kõrgemat järku funktsioon, on funktsioonitüüpi argumentide konkreetne väärtus üsna ebaoluline, kui arvutuse loogika (milliseid harusid millises järjekorras kasutatakse jms) argumentfunktsiooni valikust ei sõltu.

Näiteks operaatori `map` rakendamisel ei sõltu arvutuse käik argumentfunktsioonist: alati rakendatakse argumentfunktsioon järjest argumentlisti igale elemendile.

Sel juhul võib kõrgemat järku funktsiooni testimisel piirduda tema funktsioonitüüpi argumendi üheainsa keerulisema väärtusega; korrektsus selle korral annab kindluse, et arvutus on õige ka teistel funktsioonidel. Nii lihtsustab koodi korduvkasutus ka testimist. Ühe arvutusskeemiga võib leida ka sama ülesande eri täpsusastmega lahendusi, mis võivad üksteise korrektsust kinnitada.

Kombinatoorikas kohtab tihti ülesandeid leida mingi omadusega objektide arv. Kuid tihti õnnestub väljendada nii objektide nimekiri kui nende arv sama polümorfse operaatori kaudu, mille väärtus on kõrgemat järku funktsioon. Näiteks sellest oli juba bitisõnede arvutamine koodiga (193) (bitisõnede arv 2^n ilmselt avaldub kõrgemat järku astendamise kaudu).

Sama kõrgemat järku funktsiooni kasutamine nii arvude kui ka nimekirjade avaldamiseks annab teatava garantii, et leitud arvud tõepoolest vastavad lahendatud kombinatoorsele ülesandele. See on tähtis, sest arve saab arvutada parameetri palju suuremate väärtuste jaoks kui objektide liste, mis võivad olla pikad. Naiivne variant defineerida arvud lihtsalt nimekirja pikkusena annaks sama garantii ainult nende juhtude jaoks, millele suudetakse kogu nimekiri valmis genereerida.

Järgnevas läbi tehtav näide käsitleb Catalani arve. Seostega (134) matemaatiliselt defineeritud ja deklaratsioonidega (135), (136), (137) Haskellis kodeeritud Catalani jada on tuntud oma paljude kombinatoorsete interpretatsioonide poolest.

Üks kuulsamaid on kindla pikkusega nn sulusõnede loendamine: sõnesid, mis koosnevad (üht tüüpi) avavatest ja lõpetavatest sulgudest ning kus neid on para-

jasti n paari ja nad on tasakaalus, on täpselt c_n tükki. Näiteks $n = 3$ puhul on 5 sulusõnet “()()()”, “()(())”, “(())()”, “(()())”, “((()))”.

Samad arvud ilmnevad ka näiteks järgmises loendamisülesandes. Nimetame permutatsiooni $(a_1, \ldots, a_n)$ arvudest 1 kuni n **liibuvaks**, kui alati, kui mingite indeksite i, j korral $i < j$, kuid $a_i > a_j$, siis kõik suuruselt a_j ja a_i vahele jäävad arvud esinevad permutatsioonis enne a_j. Osutub, et liibuvaid permutatsioone arvudest 1 kuni n on täpselt c_n.

Kokkulangevus tuleneb sellest, et loendatavate objektide nimekiri kombinatoorses ülesandes avaldub sama ülesande objektide nimekirjade kaudu parameetri väiksemate väärtuste korral sama rekursiooniskeemiga nagu Catalani arvud. Seetõttu kui abstraheerime Catalani arve arvutava koodi sobivalt, võime uute rakendustena lisaks arvudele saada kätte mitme kombinatoorse ülesande objektide nimekirjad.

See, et objektide loetelu arvutamine võib toimuda sama skeemi järgi nagu nende arvu arvutamine, ei peakski väga üllatama. Kombinatoorsete objektide arvu valemil on konkreetsed põhjused, ta kajastab objektide täieliku läbivaatuse skeemi.

Kus täielik läbivaatus leiab ühe ja teise loetelu objektide kõikvõimalikud paarid, seal tuleb arvu leidmiseks korrutada; kus täielikul läbivaatusel ühendatakse ühisosata loetelud üheks, seal tuleb arvu leidmiseks liita. Kombinatoorikas tuntakse neid seaduspärasid vastavalt **korrutamisreegli** ja **liitmisreegli** nime all. Üksikule triviaalsele objektile täielikul läbivaatusel vastab arv 1 loendamisel. Osutub, et abstraheerimine korrutamisest, liitmisest ja ühikust on antud ülesande puhul piisav.

Muutuja `liidaVastavad` definitsioonis (135) on vaja abstraheerida vaid liitmisest (korrutamine ega ühik siin ei osale). Esimene mõte on listide vastavate elementide kokkuarvutamiseks kasutada operaatorit `zipWith`; see siiski ei sobi, sest antud juhul ei tohi pikema listi ülejäävat sabaosa ära lõigata, sealsed elemendid peavad minema tulemusse (listi lõppemine tähendab, et ülejäänud kordajad on nullid, ja nullide liitmisel teise listi kordajatele jäävad nood alles).

Seepärast kirjutame sarnase operaatori `zipWith'`, mille puhul pikema listi ülejääv sabaosa jääks tulemuslisti lõppu, koodiga

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

zipWith' (+) (a : as) (b : bs)
  = a + b : zipWith' (+) as bs
zipWith' _   xs       []
  = xs
zipWith' _   _        ys
  = ys
```

Kõrgemat järku konvolutsioon peab võtma argumendiks nii liitmise kui ka korrutamise. Rohujuuretasandi konvolutsiooni definitsiooni (136) neist abstraheerides saa-

me väga üldise definitsiooni

```
kõrgKonv (+) (*) xs ys
  = let
      xs@ ~(a : as) *#* ys@ ~(b : bs)
        | null xs || null ys
          = []
        | otherwise                                          ,
          = a * b :
            zipWith' (+) (map (a *) bs) (as *#* ys)
    in
    xs *#* ys
```

(194)

kus *let*-avaldis on kasutusele võetud selleks, et võimalikult palju varasemat koodi õnnestuks muutmata üle kanda. Uue definitsiooni üldisust näitab signatuur

```
kõrgKonv
  :: (c -> c -> c) ->
     (a -> b -> c) ->  '
     [a] -> [b] -> [c]
```

kust ilmneb, et korrutamine ei pea olema operatsioon ühel tüübil, vaid kumbki argumenditüüp ja väärtusetüüp võivad lausa kõik erinevad olla.

Catalani jada definitsiooni (137) üldistus võtab lisaks argumendiks ka ühiku, millega list algab; saame signatuuri

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

koos definitsiooniga

```
eneseKonv (+) (*) e
  = let
      cs
        = e : kõrgKonv (+) (*) cs cs  '
    in
    cs
```

Edasi võib välja mõelda liitmisele ja korrutamisele vastavad operatsioonid ning ühiku näiteks sulusõnede listide peal. Liitmise kui loetelude ühendamise kohale on lihtsaim valik konkatenatsioon (kui meil oleksid erisoovid järjekorra osas, siis võibolla peaksime midagi muud valima). Ühik, lähtuvalt `eneseKonv` definitsioonist, läheb kohale 0, seega ta peab olema list kõigist 0 sulupaariga sulusõnedest ehk list, mille

ainus element on tühi sõne. Kahe võimaluste listi korrutamisel võetakse ühest ja teisest listist kõikvõimalikel viisidel elementide paar ja kombineeritakse omavahel mingil kindlal viisil. Sellist operatsiooni me juba nägime bitisõnede listi moodustamisel ja ta sai defineeritud koodiga (192) muutuja `otsekorruta` väärtuseks parameetrilisena kombineerimisviisi suhtes.

Vajaliku kombineerimisviisi väljaselgitamiseks uurime antud kombinatoorset ülesannet lähemalt; peame nägema vastavust ülesande ja selle arvutusskeemi vahel, mida üritame kasutada. Olgu meil n sulupaari, $n \geqslant 1$. Sulusõnes peab esimene sümbol olema avav sulg, muidu poleks ta tasakaalus. See sulg koos talle vastava lõpetava suluga jagab sulusõne kaheks väiksemaks sulusõneks: üks jääb vaadeldud sulgude vahele, teine järgneb lõpetavale sulule. Neis on kokku $n - 1$ sulupaari ja sulupaaride jagunemine ühe ja teise osa vahel on suvaline. Kuna sulud on üht tüüpi, siis loeb ainult arvuline jagunemine, st valides ühte poolde i sulupaari, jääb neid teise $n - 1 - i$, ja nii iga i jaoks, kus $0 \leqslant i \leqslant n - 1$.

Siit tulenebki konvolutsioonilaadne skeem, mida nägime Catalani arvude definitsioonis (134). Seega kombineeruvad just esimest sulgu sisaldava sulupaariga ümbritsetud ja sellest väljajääv osa. Järelikult kombineerimiseks tuleb esimene sulusõne ümbritseda sulgudega ja paremale lisada teine sulusõne. Selle realiseerib kood

```
suluKomb as bs
  = '(' : as ++ ')' : bs'
```

vastav signatuur on

```
suluKomb
  :: String -> String -> String'
```

Kokkuvõttes saame, et listi, mille elementideks on kõikvõimalike sama sulupaaride arvuga sulusõnede listid sulupaaride arvu kasvamise järjekorras, võib kirjeldada koodiga

```
sulusõned
  :: [[String]]'

sulusõned
  = eneseKonv (++) (otsekorruta suluKomb) [""]'
```

Kindla arvuga sulusõnede loetelu leidmiseks piisab sellest listist vastava numbriga element välja võtta.

Teeme sama ka liibuvate permutatsioonide jaoks. Nullist arvust leidub täpselt üks permutatsioon — tühi — ja see on liibuv. Kui $n \geqslant 1$, siis iga liibuva permutatsiooni arvudest 1 kuni n jaotab kaheks osaks arv 1. Osade pikkused võivad olla suvalised, kuid nende summa on $n - 1$.

Kui vasakpoolse osa pikkus on i, siis ta sisaldab parajasti arvud 2 kuni $i + 1$, sest mõne arvu vahelejätmine oleks liibuvusega vastuolus. Seejuures lahutades igast selle osa arvust 1, saame liibuva permutatsiooni arvudest 1 kuni i. Parempoolne osa sisaldab arvud $i + 2$ kuni n ja kui kõigist lahutada $i + 1$, saame liibuva permutatsiooni arvudest 1 kuni $n - 1 - i$. Teiselt poolt, pannes suvaliselt paika arvu 1 ja täites vasakpoolse ja parempoolse osa nii, et vastavalt 1 ja $i + 1$ lahutamisel osa kõigist komponentidest tekiksid liibuvad permutatsioonid, on tulemuseks alati liibuv permutatsioon arvudest 1 kuni $n + 1$.

Järelikult liibuvate permutatsioonide ülesandes tuleb kahe väikse liibuva permutatsiooni kombineerimiseks koostada permutatsioon, mille algusosa tuleb esimesest permutatsioonist, kus komponentidele on liidetud 1, järgneb arv 1 ja lõpuosa tuleb teisest permutatsioonist, mille komponentidele on liidetud $i + 1$, kus i on algusosa pikkus. Esitades permutatsioone lühikeste täisarvude listina, saame signatuuri ja definitsiooni

```
liibuvKomb
  :: [Int] -> [Int] -> [Int]'

liibuvKomb as bs
  = let
      komb i (a : as)
        = a + 1 : komb (i + 1) as
      komb i _                        .
        = 1 : map (i +) bs
    in
    komb 1 as
```
(195)

(Lisaparameetris `i` arvutatakse esimese osa pikkus.)

Lõpmatu listi, mille element kohal n on list kõikvõimalikest liibuvatest permutatsioonidest arvudest 1 kuni n, on siis kirjeldatav koodiga

```
liibuvad
  :: [[[Int]]]
liibuvad                                                    .
  = eneseKonv (++) (otsekorruta liibuvKomb) [[]]
```
(196)

Muidugi saame ka Catalani arvude listi enda sama kõrgemat järku funktsiooni rakendusena kätte, definitsioon oleks

```
cats
  = eneseKonv (+) (*) 1.
```
(197)

Kõiki sulusõnesid või liibuvaid permutatsioone genereeriv arvutus muutub talumatult aeglaseks juba teises kümnes, samas kui Catalani arve saab koodiga (197) arvutada tuhandeid. Pärinemine sama kõrgemat järku funktsiooni rakendusest annab

kindluse, et leitud arvud on õiged ka neil juhtudel, kus sulusõnede või liibuvate permutatsioonide täisnimekirja genereerida ei jõua.

Ülesandeid

408. Modifitseerida muutuja `liibuvKomb` definitsiooni koodis (195) nii, et koodiga (196) defineeritud muutuja `liibuvad` väärtuseks oleva listi element kohal n väljendaks nimekirja kõigist n komponendiga järjenditest, mis on mittekahanevad ja mille ükski komponent pole väiksem oma järjekorranumbrist (lugedes 1-st) ega suurem n-st.

409. Vaatame järgmist kombinatoorikaülesannet. Ringjoon on $2n$ punktiga jaotatud võrdseteks kaarteks; nummerdame jaotuspunktid järjekorras arvudega 0 kuni $2n - 1$. Komplekti n kõõlust antud punktide vahel, kus ükski kaks ei lõiku ega oma ühist otspunkti, nimetame **kõõlude vabasüsteemiks**. Kuna igas kõõlude vabasüsteemis ühendab iga kõõl eri paarsusega punkte (muidu jääks otspunktide vahele paaritu arv jaotuspunkte, mida enam ei saaks kõõlude vahel täpselt jagada), siis võime kõõlude vabasüsteemi kirjeldada listiga, milles paarisarvude kasvamise järjekorras on nendest algavate kõõlude teiste otspunktide numbrid.

   Defineerida muutuja `kõõludKomb` nii, et definitsiooniga

```
kõõlud
  = eneseKonv (++) (otsekorruta kõõludKomb) [[]]
```

   saab muutuja `kõõlud` väärtuseks list, mille element kohal n on list kõigist kõõlude vabasüsteemidest selle n jaoks.

410. Vastavuses tüübisignatuuridega

```
üldPerm
  :: (Integer -> a -> a) -> a -> [a]'
permKorruta
  :: Integer -> [[Int]] -> [[Int]]'
permÜhik
  :: [[Int]]
```

   defineerida muutujad `üldPerm`, `permKorruta`, `permÜhik` nii, et avaldise `üldPerm permKorruta permÜhik` väärtuseks olevas listis kohal n asub list kõigist arvude 1 kuni n permutatsioonidest, samas kui definitsioon

```
facs
  = üldPerm (*) 1
```

   lahendab ülesande 246.

5.3.3 Universaalne rekursioon

Oleme näinud, et rekursiooniskeemide abstraheerimisega saab nende korduva realiseerimise mitmesuguste funktsioonide definitsioonides asendada mingi operaatori väljakutsetega ühtedel või teistel argumentidel. Vaatleme seda eesmärki korraks kui asja iseeneses ja küsime, mil määral üldse on rekursiooni peitmine funktsioonidesse võimalik.

1. Üleminek püsipunktivõrrandile.

1.1. Püsipunktivõrrand. Abstraheerimise piiride kompimisel pakuvad erilist huvi definitsioonid, mis koosnevad ühest deklaratsioonist kujul

$$\mathfrak{x} \;=\; \mathfrak{f}\ \mathfrak{x}, \tag{198}$$

kus $\mathfrak{f}$ on mingi avaldis, mis defineeritavaid muutujaid ei sisalda (vähemalt mitte defineeritavas tähenduses) ja mille koostamisel pole rekursiooni vaja.

Koodist (198) nähtub, et tüübikorrektsuseks peab avaldise $\mathfrak{f}$ tüüp olema funktsioonitüüp, kus argumenditüüp ja väärtusetüüp langevad kokku — mõlemad võrduvad $\mathfrak{x}$ tüübiga.

Matemaatikas nimetatakse funktsiooni f argumenti x, mille korral $f\ x = x$, tema **püsipunktiks**. Sellest lähtuvalt nimetatakse kuju (198) **püsipunktivõrrandiks**: ta sätestab, et $\mathfrak{x}$ väärtus on $\mathfrak{f}$ väärtuse püsipunkt.

Kuju (198) on n-ö minimalistlik rekursiooniskeem. Funktsioonide defineerimisel on selline rekursiooniskeem praktikas ebatüüpiline, kuna defineeritav $\mathfrak{x}$ esineb paremal ainult argumendi positsioonis. See tähendab, et otsest rekursiivset pöördumist ei ole, $\mathfrak{x}$ uue väljakutse tekkimiseks peab $\mathfrak{f}\ \mathfrak{x}$ arvutus $\mathfrak{x}$ välja kutsuma, mistõttu uue väljakutse asjaolud sõltuvad avaldisest $\mathfrak{f}$. Kuju (198) on loomulikum listide rekursiivsel defineerimisel.

Kõigest hoolimata osutub, et kõik rekursiivsed definitsioonid on võimalik samaväärselt püsipunktivõrrandi kujul (198) ümber kirjutada.

1.2. Lihtsamad definitsioonid. Iga otseselt rekursiivne definitsioon esitab defineeritava andme tema enda kaudu.

Lihtsaimal juhul on definitsioon kujul

$$\mathfrak{x} \;=\; \mathfrak{e}, \tag{199}$$

kus $\mathfrak{x}$ on muutuja ja $\mathfrak{e}$ on avaldis. Avaldist $\mathfrak{e}$ muutujast $\mathfrak{x}$ abstraheerides tekibki kuju (198) jaoks sobiv $\mathfrak{f}$. Vastavalt konstruktsioonile on $\mathfrak{f}\ \mathfrak{x}$ avaldisega $\mathfrak{e}$ samaväärne.

Lõpmatu nullide listi definitsiooni (90) vasak pool on muutuja `zs` ning parem pool on `0 : zs`. Seega tuleb avaldis `0 : zs` abstraheerida muutujast `zs`, millega saame endisega samaväärse püsipunktivõrrandi kujul definitsiooni

```
zs
  = (0 :) zs .
```
(200)

Kui püsipunktivõrrandis (198) panna f kohale mehhaanilisel abstraheerimisel tekkiv lambdaavaldis `\ x -> e`, muutuks definitsioon kohmakaks ja halvasti jälgitavaks. Seetõttu tasub püsipunktivõrrandi (198) f kohale pigem võtta uus muutuja, mis eraldi defineeritud samaväärseks vajaliku lambdaavaldisega. Sobib funktsionaalse vasaku poolega deklaratsioon

```
f x
  = e .
```
(201)

Märkame, et deklaratsioon (201) saadakse originaaldeklaratsioonist (199) uue muutuja f mehhaanilise lisamisega vasaku poole algusse. Kuna lisatav muutuja on uus, pole nii saadud definitsioon kunagi rekursiivne.

Sellise lükke sobivuses võib ka vahetult veenduda: muutuja x väärtustamisel püsipunktivõrrandi (198) järgi tekib vahetulemus $f\ x$, mis deklaratsiooni (201) järgi annab järgmiseks vahetulemuseks e. Võrreldes väärtustamisega originaaldefinitsiooni (199) järgi on lihtsalt üks vahesamm juures.

Vaatame geomeetrilise jada definitsioonis (89) olevat lokaalset deklaratsiooni, mis defineerib rekursiivselt muutuja `gs`. Lisades sinna vasaku poole algusse uue muutuja `abstr` ja tuues sisse püsipunktivõrrandi, saame koodiga (89) samaväärse koodi

```
geom a q
  = let
      abstr gs
        = a : [x * q | x <- gs]
      gs
        = abstr gs
    in
    gs
```
, (202)

kus ainus rekursiivne definitsioon on püsipunktivõrrandi kujul (198).

1.3. Deklaratsioonisüsteemiga definitsioonid. Kui muutujat x defineerib üks funktsionaalse vasaku poolega deklaratsioon, on see samaväärne deklaratsiooniga kujul (199), kus paremal pool on lambdaavaldis. Abstraheerimisel muutujast x tekib lambdaavaldise algusse formaalne parameeter x juurde.

Nagu varemgi, tasub püsipunktivõrrandi (198) kasutamiseks võtta f kohale uus muutuja, mille väärtus võrduks selle lambdaavaldise omaga. Selles definitsioonis võib viia lisaks muutujale x vasakule poole ka kõik need parameetrid, mis algselt juba olid seal. Sellega tekib deklaratsioon, mille saaksime algsest ka otse juba tuttaval moel, lisades vasaku poole algusse mehhaaniliselt uue muutuja, mis lükkab defineeritava muutuja x formaalse parameetri kohale.

Püsipunktivõrrand (198) osutub siis definitsiooniks argumendivabas stiilis. Lisades mõlemale poole vajaliku hulga argumente, saab analoogselt ülal vaadeldud paljast muutujast koosneva vasaku poolega juhuga veenduda, et arvutus püsipunktivõrrandi kaudu annab kahe sammuga sama vahetulemuse mis originaaldefinitsioon ühega.

Näiteks võtame esimese selles õpikus vaadeldud rekursiivse definitsiooni (63), mis andis muutuja `suurSumma` väärtuseks naturaalarvude 4. astmete summasid leidva funktsiooni. Lisades vasakusse poolde uue operaatori `suurSummaAbstr`, mis lükkab muutuja `suurSumma` selle argumendiks, saame koodi

```
suurSummaAbstr suurSumma n
  = case compare n 0 of
      GT
        -> suurSumma (n - 1) + n ^ 4
      EQ
        -> 0
      _
        -> error "suurSumma: neg. liidetavate arv"
```
(203)

Kuna endise `suurSumma` tüüp oli `(Integral a) => a -> a`, siis operaatori `suurSummaAbstr` signatuuriks sobib

```
suurSummaAbstr
  :: (Integral a)
  => (a -> a) -> a -> a
```

Muutuja `suurSumma` saame aga endisega samaväärselt defineerida püsipunktivõrrandi (198) kujul deklaratsiooniga

```
suurSumma
  = suurSummaAbstr suurSumma
```
(204)

Väärtustades nüüd avaldise kujul `suurSumma` a, tekib (204) järgi esimene vahetulemus `suurSummaAbstr suurSumma` a, mis (203) järgi annab järgmisena

sama vahetulemuse mis `suurSumma α` oleks originaaldefinitsiooni (63) järgi andnud esimesel sammul.

Kui definitsioon sisaldab rohkem kui ühe deklaratsiooni, on vastava abstraktsiooni defineerimine teostatav sama lükkega kui seni, lihtsalt üks ja sama uus muutuja tuleb lisada igasse deklaratsiooni. Sama töötab ka juhul, kui mõni parem pool on valvurikonstruktsioon. Selles on lihtne veenduda, vaadeldes väärtustamist konkreetsete argumentide korral või siis kasutades asjaolu, et mainitud hargnemised on ümber kirjutatavad valikuavaldisega.

Läheme püsipunktivõrrandile üle ka kahe järjestatud listi põimimise operaatori `põim` definitsioonist (124). Lisades uue muutuja `põimAbstr` kõigi deklaratsioonide algusse, saame koodi

```
põimAbstr põim xs@ (a : as) ys@ (b : bs)
  | a <= b
    = a : põim as ys
  | otherwise
    = b : põim xs bs                    ,        (205)
põimAbstr põim xs           []
  = xs
põimAbstr põim _            ys
  = ys
```

mis tõrjub muutuja `põim` formaalseks parameetriks. Kuna endise `põim` tüüp oli `(Ord a) => [a] -> [a] -> [a]`, siis `põimAbstr` signatuuriks sobib

```
põimAbstr
  :: (Ord a)                                        .
  => ([a] -> [a] -> [a]) -> [a] -> [a] -> [a]
```

Edasi saame kirjutada püsipunktivõrrandi (198) kujul deklaratsiooni

```
põim                                                (206)
  = põimAbstr põim'
```

mis annab `põim` väärtuseks sama funktsiooni mis originaaldefinitsioon (124).

Väärtustades avaldist kujul `põim` l_1 l_2 definitsiooni (206) järgi, saame vahetulemuse `põimAbstr põim` l_1 l_2, mis definitsiooni (205) järgi omakorda annab sõltumata harust sama vahetulemuse mis oleks tekkinud algse avaldise väärtustamisel originaaldefinitsiooni (124) järgi.

1.4. Muud definitsioonid. Deklaratsiooni vasak pool võib olla paar või mõni muu muutujast keerulisem näidis; definitsioon võib seejuures olla rekursiivne.

Sellist olukorda nägime koodis (95).

Ka sel juhul saab tuttava mehhaanilise lükkega abstraheerida ja üle minna püsipunktivõrrandile kujul (198). Abstraktsioon toimub mitte ühest muutujast, vaid näidisest, mis on sisult samaväärne abstraheerimisega kõigist näidises esinevatest muutujatest.

Koodis (95) annaks püsipunktivõrrandile üleminek tulemuseks

```
järgmVeerg (u , v)
  = let
      pAbstr p@ ~(s , _)
        = (u + v , u + s) .                    (207)
      p = pAbstr p
    in
    p
```

Kuna tulemuses pole muutujat `s` enam vaja, siis on teises lokaalses deklaratsioonis näidisest paar ära jäetud. Teiseks muutuseks on paarinäidise laisakssundimine esimeses lokaalses deklaratsioonis, et säilitada sisulist vastavust originaaldefinitsiooniga.

1.5. Kaudse rekursiooniga definitsioonisüsteemid. Senised võtted tagasid edu otsese rekursiooni korral, mis ei tekitanud defineeritavate muutujate kaudset sõltuvust iseendast läbi teiste muutujate.

Mitme definitsiooni kaudse sõltuvuse korral üksteisest tuleb selliste definitsioonide plokki käsitleda kui üht definitsiooni, mis korraga määratleb kõik nende poolt defineeritavad muutujad. Ühe definitsioonina kirjutamiseks tuleb kõik need muutujad paigutada andmestruktuuri. Edasi tuleb talitada samamoodi nagu varem selliste definitsioonide puhul, kus vasak pool oli muutujast keerulisem näidis.

Selgituseks töötame püsipunktivõrrandi kujule ümber muutujate `kaheksLoe` ja `kaheksLoe'` defineeriva koodi (80), kus definitsioonid on vastastikku rekursiivsed. Püsipunktina saame väljendada nende muutujate väärtuseks olevate funktsioonide paari.

Abstraheeriva operaatori definitsioon võib välja näha kujul

```
kaheksLoeAbstr ~(kaheksLoe , kaheksLoe')
  = let
      k (x : xs)
        = let
            (us , vs)
              = kaheksLoe' xs
          in
          (x : us , vs)
      k _
        = ([] , [])
      k' (x : xs)
        = let
            (us , vs)
              = kaheksLoe xs
          in
          (us , x : vs)
      k' _
        = ([] , [])
    in
    (k , k')
```

Tema väärtus on funktsioon, mis teisendab funktsioonide paare funktsioonide paarideks, kus funktsioonid paarides on sama tüüpi nagu endise `kaheksLoe` väärtus. Signatuuriks sobib

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

Järgneb püsipunktivõrrandi kujul definitsioon. Kuna lõppeesmärk on defineerida `kaheksLoe`, siis teise komponendi kohal on jokker. Sobib kood

```
p@ (kaheksLoe , _)
  = kaheksLoeAbstr p
```
(208)

Ülesandeid

411. Kirjutada samaväärselt püsipunktivõrrandi kaudu ümber definitsioonid (72), (73), (86).

412. Kirjutada samaväärselt ümber muutuja `üleÜheEtte` definitsioon (96) ja muutuja `sumSees` definitsioon (97), nii et rekursioon oleks viidud püsipunktivõrrandi kujule.

2. Püsipunktikombinaator.

2.1. Defineerimine. On jäänud teha veel üks abstraktsioon, mis toob välja parajasti kõigi kujul (198) olevate definitsioonide ühise struktuuri.

Paneme tähele, et kuju (198) sõltub reaalselt ainult avaldisest `f`, sest `x` on defineeritav üksus, mida võib nagunii tähistada kuidas iganes. Seega tuleb just `f` muuta uue üldise operaatori parameetriks, temast tuleb suvalist püsipunktivõrrandi kujul definitsiooni abstraheerida.

Defineeritav üldine operaator on klassikaline, seepärast kasutame ka tema tuntud klassikalist nime `fix`. Otse kuju (198) järgi kirjutades tekib kood

```
fix f
  = f (fix f)'
```
(209)

Muutuja `fix` väärtuseks saab funktsioon, mis seab argumentfunktsioonile vastavusse tema püsipunkti, mistõttu teda nimetatakse **püsipunktikombinaatoriks**. Sobib tüübisignatuur

```
fix
  :: (a -> a) -> a'
```

2.2. Kasutamine. Nüüd saab kõik definitsioonid kujul (198) samaväärselt ümber kirjutada operaatori `fix` rakenduse kaudu. Saadavad definitsioonid ei ole rekursiivsed.

Kirjutades näiteks definitsiooni (200) samaväärselt ümber operaatori `fix` abil, saame uue definitsiooni

```
zs
  = fix (0 :)'
```

mis pole rekursiivne, kuna `zs` ei esine paremas pooles.

Definitsioonist (202), mis andis muutuja `geom` väärtuseks esimese liikme ja teguri järgi lõpmatut geomeetrilist jada välja andva funktsiooni, saame samamoodi koodi

```
geom a q
  = let
      abstr gs
        = a : [x * q | x <- gs] '
    in
    fix abstr
```
(210)

kus on õnnestunud isegi vabaneda ühest lokaalsest deklaratsioonist. Rekursiivselt siin midagi ei defineerita.

Operaatori `suurSumma` definitsioonist (204) saame analoogselt

```
suurSumma
  = fix suurSummaAbstr'
```
(211)

Operaatori `suurSummaAbstr` definitsioon (203) oleks loomulik üle viia definitsiooni (211) kehasse *let*-plokki, sest mujal pole teda mõtet välja kutsuda (see ühtlasi garanteeriks veateate adekvaatsuse, sest veateates sai säilitatud operaatori `suurSumma` nimi).

Operaatori `kaheksLoe` definitsioonist (208) saame püsipunktikombinaatori abil definitsiooni

```
(kaheksLoe , _)
  = fix kaheksLoeAbstr'
```

kus üleminek võimaldas vabaneda ehknäidisest.

2.3. Tähtsus. Oleme näinud, et iga rekursiivse definitsiooni saab samaväärselt ümber kirjutada nii, et rekursioon on püsipunktivõrrandi kujul, ning kõik sellised omakorda on samaväärselt rekursioonita ümber kirjutatavad `fix` kaudu. Järelikult on samaväärselt rekursioonita ümber kirjutatavad kõik rekursiivsed definitsioonid — peale ühe, mis defineerib püsipunktikombinaatori.

Püsipunktivõrrandi ja -kombinaatori tähtsus seisnebki selles, et püsipunktikombinaator on ainuke operaator, mille defineerimiseks tegelikult rekursiooni vaja on. Defineerides püsipunktikombinaatori, oleme rekursiooni kui sellise välja abstraheerinud.

On mõeldav Haskelliga väljendusvõimsuselt võrdne funktsionaalne keel, milles rekursiivsed definitsioonid pole lubatud, kuid milles on püsipunktikombinaator sisse ehitatud.

2.4. Optimeerimine. Andmestruktuuride puhul võib juhtuda, et arvutus `fix` definitsiooni (209) kasutava koodiga on ebaefektiivne võrreldes arvutusega püsipunktivõrrandi kujul (198) oleva definitsiooni järgi.

Katsetamine näitab, et geomeetrilise jada arvutus definitsioonide (209), (210) järgi on palju aeglasem kui arvutus definitsiooni (202) järgi.

Aegluse põhjus peitub `fix` ebaõnnestunud definitsioonis (209): iga rekursiivne pöördumine `fix` poole arvutab `fix f` tulemust omaette ja tekivad tarbetud kopeerivad arvutused.

Meenutagem definitsiooni (89) saamislugu varasemast samaväärsest, kuid aeglasemast definitsioonist (88). Üleminekul funktsioonirekursioonilt listirekursioonile kiirus kasvas: kui enne arvutas iga rekursiivne pöördumine tulemuslisti otsast peale, siis tulemuslisti sidumisel muutujaga arvutati teda vaid ühe korra välja.

Seepärast on vaja operaatori `fix` definitsioonis siduda väljaantav väärtus eraldi lokaalse muutujaga. Saame koodiga (209) samaväärse definitsiooni

```
fix f
  = let
      x = f x .                                    (212)
    in
    x
```

Definitsiooni (212) võib saada kohe püsipunktivõrrandi kujul oleva koodi abstraheerimisel ilma vaheetapita (209), nii nagu näiteks konvolutsiooni definitsioonist (136) tekkis kõrgemat järku konvolutsioon (194).

Definitsiooni (212) korral töötab arvutus definitsiooni (210) järgi praktiliselt niisama kiiresti kui definitsiooni (202) või originaaldefinitsiooni (89) järgi.

Definitsioon (212) on kasutatav (209) asemel mitte ainult andmestruktuuride, vaid ükskõik millist tüüpi andmete kirjeldamiseks. Kui varem võisime arvet pidada, kas tegu on funktsiooni-, protseduuri- või listirekursiooniga, siis koodi (212) *let*-plokis defineeritakse rekursiivselt muutuja `x`, mille tüüp polegi teada ja võib konkreetsetel rakendusel olla mis iganes. Juhul, kui kirjeldatakse funktsiooni, on see lokaalne definitsioon argumendivabas stiilis.

Muidugi pole püsipunktikombinaator sisukalt rakendatav suvalisele tüübilt sobivale argumendile.

Näiteks `fix sin` väärtustamine jääb lõpmatult tööle.

Avaldise `fix` ƒ väärtustamine jääb lõpmatult tööle või lõpetab täitmisaegse veaga alati, kui ƒ väärtus on agar funktsioon. Teooria klapib, sest bottom on iga agara funktsiooni püsipunkt (agarus just seda tähendabki).

Ülesandeid

413. Kirjutada püsipunktikombinaatoriga samaväärselt ümber muutujate `põim` ja `järgmVeerg` definitsioonid (206) ja (207).

414. Kirjutada avaldis kujul `fix` ƒ, kus ƒ tüüp on `Int -> Int` ja mille väärtustamine lõpeb normaalselt.

6 Maailma laiendamine

6.1 Klassid

Oleme pidevalt puutunud kokku polümorfismiga. Haskelli iga polümorfse süntaktilise üksuse kõik tüübid on esitatavad tüübiparameetrite abil ühtsel viisil. Seda omadust nimetatakse parameetriliseks polümorfismiks.

Mitte ainult polümorfse avaldise tüüp, vaid ka tema väärtus on parameetriline nendesamade tüübiparameetrite suhtes, kuna polümorfsel avaldisel on üks väärtus igast oma tüübist. Tüübiparameetreid kirjutatakse aga ainult tüübiavaldistesse, mujal jäävad nad ilmutamata.

Näiteks operaatori `map` tüüp on `(a -> b) -> [a] -> [b]`, mis sisaldab kaht tüübimuutujat. See tähendab, et operaatoril `map` on lisaks funktsioonile ja listile veel ka kaks tüübiargumenti, mida `map` kasutamisel ei kirjutata. Näiteks avaldises `map (== 'X')` on ühe tüübiparameetri väärtus kindlasti Char ja teine Bool.

Kuna tüübiargumendid määravad ülejäänud argumentide tüübi, siis tüübiargumendid põhimõtteliselt koguni eelnevad ülejäänud argumentidele.

Avaldist nimetatakse universaalselt polümorfseks, kui ta võtab küll väärtusi mitmest tüübist, kuid temaga ümberkäimine tüübist ei sõltu. Universaalne polümorfism vastandub ülelaadimisele, kus polümorfse avaldise väärtustamine kulgeb sõltuvalt tema tüübist üht- või teistviisi.

Formaalselt eristab ülelaadimist universaalsest polümorfismist tüübikontekst signatuuris — ülelaadimine ei saa kunagi toimuda üle kõigi tüüpide, vaid ainult üle klassidega piiratud tüübiperede.

Operaator `map` on näiteks universaalsest polümorfismist. Üle laetud on aga näiteks aritmeetilised operatsioonid, sest nende sooritamise viis sõltub sellest, millist laadi arvudega on tegu.

Universaalselt polümorfsete muutujate defineerimise juures pole mingit kunsti, sest seal programmeeritav arvutus tüübist ei sõltu ja tüüp definitsioonis üldse ei esine.

Selliseid definitsioone oleme näinud ja kirjutanud juba küll ja küll.

Sama olukord on üle laetud muutujatega, kui nad defineerida olemasolevate üle laetud operaatorite kaudu ühtsel viisil. Siis ülelaadimine kandub olemasolevatelt uutele muutujatele automaatselt üle.

Sellised on kõik eelnevas nähtud definitsioonid, mis annavad muutuja väärtuseks klassiga kitsendatud väärtuste pere, st signatuuris esineb tüübikontekst.

Teine lugu on ülelaadimise sissetoomisega otse "õhust". Loogiliselt võttes peaks olema tegemist hargnemiskonstruktsiooniga, kus haru valik toimub tüübi järgi.

Sellise "konstruktsiooni" ülesehitus aga erineb tavalistest hargnemiskonstruktsioonidest kardinaalselt. Erinevate tüüpide kohta käivad juhud on üle kogu koodi laiali ja programmeerija saab juhte meelevaldselt oma moodulites juurde lisada, laiendades vajadusel ka standardteegist või muust mittemuudetavast teegist tulevate üle laetud muutujate ja konstruktorite kasutusvõimalusi.

6.1.1 Klasside sissetoomine ja laiendamine

1. Klassi meetodid. Et muutujat saaks defineerida erinevate tüüpide jaoks erinevalt, peab ta olema mingi klassi meetod. Klassi meetod on muutuja, mis on klassi definitsioonis meetodina deklareeritud. Mujal seda määrata ega muuta ei saa.

Olemasolevate klasside meetodite nimekirja saab küsida interaktiivses keskkonnas käsuga `:i`.

Näiteks klassi Eq jaoks näitab Hugs, et tema meetodid on `==` ja `/=`, ja esitab ka nende muutujate tüübid. Klassi Num jaoks annab GHCi teada, et tema meetodid on `+`, `*`, `-`, `negate`, `abs`, `signum` ja `fromInteger`.

2. Klassideklaratsioonid. Klass defineeritakse klassideklaratsiooniga

kujul

class $\mathfrak{h}$ **where**

$\mathfrak{s}_1$
$\ldots$
$\mathfrak{s}_l$,

$\mathfrak{d}_1$
$\ldots$
$\mathfrak{d}_k$

kus $\mathfrak{s}_1, \ldots, \mathfrak{s}_l$ on tüübisignatuurid ning $\mathfrak{d}_1, \ldots, \mathfrak{d}_k$ definitsioonid, kusjuures $l \geqslant 0$ ja $k \geqslant 0$. Klassideklaratsiooni päises **class** $\mathfrak{h}$ **where** olev üksus $\mathfrak{h}$ on lihtsamal juhul kujul $\mathfrak{c}$ $\mathfrak{a}$, kus $\mathfrak{c}$ on uue klassi nimi ja $\mathfrak{a}$ on tüübimuutuja, mis tähistab suvalist selle klassi potentsiaalset esindajat. Seda tüübimuutujat saab ja tuleb kasutada kõigis signatuurides $\mathfrak{s}_1, \ldots, \mathfrak{s}_l$. Tingimust $\mathfrak{c}$ $\mathfrak{a}$ konteksti kirjutada ei tohi, aga ta kehtib vaikimisi. (Tüübikontekst kui selline on neis signatuurides siiski lubatud.)

Signatuuride vasakud pooled peavad olema erinevad muutujad; need saavadki klassi meetoditeks. Definitsioonid $\mathfrak{d}_1, \ldots, \mathfrak{d}_k$ on meetodite **vaikedefinitsioonid**. Vaikedefinitsioone ei pea esitama kõigi meetodite kohta, vaid vabal valikul. Iga vaikedefinitsioon võib defineerida ainult ühe meetodi.

Oleme ehk märganud, et mitmed tuttavad tüübid on sümmeetrilised, nii et neis on mõtet rääkida "vastandväärtusest": muidugi kuuluvad siia arvutüübid, kuid analoogilist rolli täidab tõeväärtuste puhul eitus ning suurusvahekorratüübi puhul vastupidise vahekorra leidmise operatsioon.

On võimalik, et erinevates olukordades, mis kasutavad mingite andmetevaheliste suhete iseloomustamiseks arve, tõeväärtusi ja võrdlusvahekordi, on kõikjal vaja kasutada suhte ümberpööramise ehk vastandväärtuse leidmise operatsiooni sarnasel viisil (näiteks listide kõigile elementidele rakendada).

Programmeerijale oleks mugav ja kood saaks ülevaatlikum, kui me ei peaks ühel tüübil kasutama üht, teisel tüübil teist operaatorit. Standardteegis aga muutuja, mis oleks kasutatav kõigis neis olukordades, puudub. Toome selle sisse klassideklaratsiooniga

```
class Sümm a where

  vastand
    :: a -> a
```
. (213)

Kood (213) defineerib klassi nimega `Sümm` ja sätestab, et sellesse kuulumiseks peab tüübil A olema defineeritud muutuja `vastand`, mille väärtus on tüüpi $A \to A$. Meie vajadustega see klapib, sest vastandväärtust otsime tõepoolest samast tüübist kust argument pärit.

Meetodi `vastand` tõeline tüüp, mida hakkavad deklaratsiooni (213) sisselugemisel näitama ka interaktiivsed keskkonnad, on `(Sümm a) => a -> a`. Tingimus `Sümm a` tuleneb signatuuri asukohast klassideklaratsioonis.

Kui deklaratsioon (213) on olemas, saab muutujat `vastand` definitsioonides juba kasutama hakata. Näiteks võib kirjutada koodi

```
vastandaKõik
  :: (Sümm a)  ,
  => [a] -> [a]

vastandaKõik
  = map vastand .
```
(214)

3. Esindajadeklaratsioonid. Klassideklaratsioon üksi ei pane ühtki tüüpi sinna kuuluma, ta tekitab vaid tühja klassi.

Koodiga (214) defineeritud operaatorit `vastandaKõik`, mis peaks argumentlisti järgi koostama vastandväärtuste listi, ei saa kuidagi testida, kuni sümmeetriliste tüüpide klass ei sisalda ühtki tüüpi.

Et klassideklaratsioon saaks täita oma eesmärki, tuleb meile vajalikud tüübid viia klassi esindajaks ja ühtlasi anda iga sellise tüübi jaoks meetodite definitsioonid.

Klassikuuluvust saab uutele tüüpidele laiendada spetsiaalsete esindajadeklaratsioonidega. Esindajadeklaratsioon on üldjuhul kujul

```
instance ɦ where

  𝔡1
  ...
  𝔡l
```
.

Siin $\mathfrak{d}_1, \ldots, \mathfrak{d}_l$ on meetodite definitsioonid, kusjuures $l \geqslant 0$. Päises **`instance`** ɦ **`where`** olev üksus ɦ on lihtsamal juhul kujul $\mathfrak{c}$ $\mathfrak{t}$, kus $\mathfrak{c}$ on klass ja $\mathfrak{t}$ on tüüp, mida tahetakse sinna klassi paigutada. Definitsioonid $\mathfrak{d}_1, \ldots, \mathfrak{d}_l$ annavad klassi $\mathfrak{c}$ meetodite tähenduse tüübi $\mathfrak{t}$ jaoks.

Signatuure esindajadeklaratsioonis olla ei või, kuid definitsioonid peavad vastama klassideklaratsioonis antud signatuuridele.

Esindajadeklaratsioonis ei pea tingimata kõiki vastava klassi meetodeid defineerima. Esindajadeklaratsioonis puuduvate definitsioonide aset täidavad klassideklaratsioonis olevad vaikedefinitsioonid ja kui ka klassideklaratsioonis meetodi definitsiooni pole, jääb meetodi väärtuseks bottom. Kui aga esindajadeklaratsioon meetodi definitsiooni annab, siis vastava tüübi jaoks kehtib selle meetodi see definitsioon isegi juhul, kui tal on klassideklaratsioonis ka vaikedefinitsioon. Teisi sõnu, vaikedefinitsioone on võimalik esindajadeklaratsioonis üle katta.

Näiteks tüübi Bool paneb klassideklaratsiooniga (213) sisse toodud sümmeetriliste tüüpide klassi kuuluma kood

```
instance Sümm Bool where

  vastand                 ’
    = not
```

kusjuures muutuja `vastand` väärtuseks saab eitus.

Analoogselt saab muutuja `vastand` tüüpide Int ja Ordering jaoks defineerida deklaratsioonidega

```
instance Sümm Int where

  vastand                 ’                    (215)
    = negate

instance Sümm Ordering where

  vastand GT
    = LT
  vastand LT                    .              (216)
    = GT
  vastand _
    = EQ
```

Nüüd on lihtne interaktiivses keskkonnas veenduda, et näiteks avaldiste

```
vastandaKõik [0 .. 4] :: [Int],
vastandaKõik (zipWith compare [1 .. 4] [2, 2.5 .. ]),
vastandaKõik (map ((> 0) . cos . (* pi)) [0 .. ])
```

väärtused on listid vastavalt elementidega

$$0, -1, -2, -3, -4; \qquad \text{GT, GT, EQ, LT;} \qquad \text{False, True, False, True, } \ldots .$$

Esimesel juhul kasutatakse väärtustamisel meetodi `vastand` definitsiooni tüübi Int jaoks, teisel juhul definitsiooni Ordering jaoks ja kolmandal juhul definitsiooni Bool jaoks. Esimeses avaldis on vajalik tüübisignatuur, sest vaikimisi loetakse täisarvkonstandid tüüpi Integer, mille jaoks esindajadeklaratsioon puudub.

Defineerime veel klassi, millesse võiksid kuuluda kõik tüübid, milles on mõtet rääkida andmete tsüklilisest nihkest ja vahekaugusest selle nihutamise mõttes.

Näiteks tüübil Bool võiksime, võttes aluseks loomuliku järjestuse False $<$ True, tuvastada, et tõeväärtusest False tõeväärtuse True saamiseks tuleb teha 1 samm, tõeväärtusest True tõeväärtuse False saamiseks aga -1 samm, muudel juhtudel 0 sammu. Sama võib teha mistahes loetelutüübi korral. Ka on ilmne, kuidas samasugust operatsiooni kirjeldada täisarvutüüpidel.

Kuna mõtleme just tsüklilist liikumist üle tüübi kõigi andmete, siis erinevate andmete vahelise kaugusena võib käsitleda ka ainult positiivset nihet. Kirjutame klassideklaratsiooni

```
class Tsükliline a where

  nihe
    :: Integer -> a -> a

  vahe
    :: a -> a -> Integer ,                    (217)

  posVahe
    :: a -> a -> Integer

  vahe
    = posVahe
```

mis võimaldab kasutada nii positiivset vahet kui lihtsalt vahet. Viimane osa sellest deklaratsioonist sätestab, et vaikimisi on tavaline vahe sama mis positiivne vahe.

Seda vaikeseost võiks programmeerida ka teisiti, viies tema signatuuri ja vaikedefinitsiooni klassideklaratsioonist välja. Signatuurile tuleb siis muidugi lisada klassikontekst. Sellega aga kaoks muutuja `vahe` klassi meetodite seast, mis tähendab, et muutujad `vahe` ja `posVahe` oleksid võrdse väärtusega klassi iga tüübi jaoks, puuduks võimalus mõnede tüüpide korral see definitsioon üle katta. Kuna sama väärtuse jaoks pole mõtet kaht muutujat kasutusele võtta, siis `vahe` ja `posVahe` eristamine on mõttekas vaid juhul, kui perspektiivis on võimalik ülekate, st nad on meetodid.

Koodiga (217) defineeritud klassi võib laiendada tüübile Bool deklaratsiooniga

```
instance Tsükliline Bool where

  nihe n x
    = x == even n

  vahe True  False
    = 1
  vahe False True
    = -1
  vahe _     _
    = 0

  posVahe x y
    = abs (vahe x y)
```
. (218)

Siin ei ole ära kasutatud, et `vahe` definitsioon on klassideklaratsioonis olemas, vaid ta on üle kaetud definitsiooniga, mis võimaldab ka negatiivseid vahesid. Kui meid ei huvitaks, kummal pool üksteisest andmed asetsevad, võiksime deklaratsiooni (218) ümber teha nii, et meetodi `vahe` definitsiooni seal ei esineks.

Antud juhul ei piisaks vaid `vahe` definitsiooni ärajätmisest, sest `posVahe` definitsioon on antud tema kaudu ja tagajärjena tekiks vastastikrekursioon klassideklaratsioonis antud definitsiooniga, mis viiks lõpmatusse tsüklisse. Ka meetod `posVahe` tuleks ümber defineerida. Sobib näiteks klassideklaratsioon

```
instance Tsükliline Bool where

  nihe n x
    = x == even n

  posVahe x y
    | x == y
      = 0
    | otherwise
      = 1
```
.

Ka fikseeritud pikkusega täisarvude tüüpi võib interpreteerida tsüklilisena, kus suurimale järgneb vähim. Võttes kasutusele abimuutuja `giga` koodiga

```
giga
  :: Integer'

giga
  = 2 ^ 30'
```

saame vajalikud operatsioonid ka tüübi Int jaoks defineerida koodiga

```
instance Tsükliline Int where

  nihe n i
    = let
        c = (n + toInteger i) `mod` (4 * giga)
      in
      fromInteger
      (if c >= 2 * giga then c - 4 * giga else c)

  posVahe i j
    = vahe i j `mod` (4 * giga)

  vahe i j
    = toInteger i - toInteger j
```

Muutuja `toInteger` on klassi Integral meetod ja tema väärtus on tüübiteisendus suvalisest täisarvulisest tüübist tüüpi Integer.

Laiendada saab ka juba olemasolevaid, teistes moodulites defineeritud klasse. Haskelli realisatsioonidega kaasas olevais teekides on klassis Num vaid arvutüübid, nii et näiteks tõeväärtusi pole võimalik arvudena käsitleda; kuid mõnikord võib programmeerija tahta sellest kitsendusest vabaneda. Siis võib ta kirjutada deklaratsiooni

```
instance Num Bool where

  (+)
    = (||)

  negate
    = not

  (*)
    = (&&)                    ,                    (219)

  abs
    = id

  signum
    = id

  fromInteger
    = (/= 0)
```

millega kuulutatakse tõeväärtusetüüp klassi Num kuuluvaks ja defineeritakse enamik selle klassi meetodeist. Definitsioon puudub operaatoril `-`, kuid klassiga Num tuleb kaasa vaikedefinitsioon, mis ütleb, et lahutamine on samaväärne vastandarvu liitmisega. See võiks siin sobida küll.

Deklaratsioon (219) võimaldab tõeväärtustega sooritada aritmeetilisi operatsioone nagu arvudegagi. Nii on nüüd tüübikorrektsed näiteks kirjutised `True + True`, `False * False`, `True - False`, `abs (2 + 3 == 5)` jne.

Töötavad isegi sellised näiliselt vigased avaldised nagu `True + 4`. Seda aktsepteeritakse tänu asjaolule, et täisarvulisi arvkonstante avaldistes interpreteeritakse operaatori `fromInteger` abil, mistõttu antud juhul on konstandi `4` väärtuseks True. Tegu on sama nähtusega, mis võimaldab kirjutada `5 + 0.2` jms.

Märgime, et klassi laiendamisel uuele tüübile jäävad meetodite prioriteet ja assotsiatiivsus kehtima ka uue tüübi jaoks.

Nii on esindajadeklaratsiooni (219) olemasolul operaatorite `+` ja `*` prioriteetide vahekord ka tõeväärtusetüübil samasugune nagu tavalistel arvudel.

3.1. Tüübipere defineerimine klassi esindajaks. Ühe esindajadeklaratsiooniga võib sama klassi esindajaks määrata terve tüüpide pere, mis on kirjeldatud tüübiparameetritega. Selleks tuleb esindajadeklaratsiooni päisesse klassi nime järele panna lokaalseid tüübimuutujaid sisaldav tüübiavaldis.

Teame, et funktsioonivääruse avaldised, nagu näiteks `sin`, annavad interaktiivses keskkonnas väärtustamisel tüübivea, sest funktsioonitüübid ei kuulu klassi Show. Võime aga kõik funktsioonitüübid sinna klassi lükata deklaratsiooniga

```
instance Show (a -> b) where

  show _                          ,                    (220)
    = "See ju funktsioon!"
```

mis defineerib funktsioonitüüpide jaoks muutuja `show` väärtuseks funktsiooni, mis annab konstantselt välja teksti “See ju funktsioon!”. Pärast seda on interaktiivse keskkonna senine algajale peavalu valmistav käitumine kui peoga pühitud; lastes väärtustada näiteks `sin`, tuleb vastuseks “See ju funktsioon!”.

Ülesandeid

415. Otsida Hugsi teegist üles klasside Eq, Enum, Show definitsioonid ja saada neist aru.

416. Viia läbi tsükliliste tüüpide klassi meetodi `vahe` muutmine tavaliseks muutujaks, millest tekstis on juttu.

417. Viia ühiktüüp sümmeetriliste tüüpide klassi, defineerides sobivalt meetodi `vastand`.

418. Viia tüüp Integer tsükliliste tüüpide klassi, defineerides sobivalt kõik vajalikud meetodid.

419. Viia suurusvahekorratüüp tsükliliste tüüpide klassi, defineerides kõik vajalikud meetodid.

420. Deklareerida tõeväärtusetüüp klassi Num esindajaks nii, et aritmeetilised operatsioonid vastaksid arvutustele *modulo* 2.

421. Deklareerida suurusvahekorratüüp klassi Num esindajaks nii, et LT vastaks negatiivsusele või -1-le, GT positiivsusele või 1-le ja EQ nullile.

422. Deklareerida sümbolitüüp klassi Num esindajaks nii, et iga sümbol oleks arvuna võrdne oma koodiga.

6.1.2 Pärimine

1. Kontekstiga klassideklaratsioonid. Et defineerida uus klass olemasolevate klasside alamklassiks, tuleb klassideklaratsiooni päisesse lisada vastav tüübikontekst.

Üldjuhul on päis kujul **`class`** `(`$\mathfrak{k}_1$ `a, ..., `$\mathfrak{k}_l$ `a) =>` $\mathfrak{c}$ `a` **`where`**, kus $\mathfrak{c}$ on uue klassi nimi ja $\mathfrak{a}$ tüübimuutuja nagu varemgi ja lisaks $\mathfrak{k}_1, \ldots, \mathfrak{k}_l$ on mingite olemasolevate klasside nimed. See tähendab, et defineeritav klass $\mathfrak{c}$ on klasside $\mathfrak{k}_1, \ldots, \mathfrak{k}_l$ (ja ainult nende) vahetu alamklass. Ühtlasi pärib klass $\mathfrak{c}$ kõik meetodid klassidest $\mathfrak{k}_1, \ldots, \mathfrak{k}_l$ koos signatuuride ja klassideklaratsioonidesse paigutatud vaikedefinitsioonidega.

Pärimine on ühene, kui see toimub vaid ühest vahetust ülemklassist, ja mitmene, kui vahetuid ülemklasse on mängus mitu. Klassi ülemklassid võivad ise olla erinevate klasside või ka mingi ühe ja sama klassi alamklassid. Keelatud on aga tsükliline pärilus.

Vaatleme uuesti koodiga (213) sisse toodud sümmeetriliste tüüpide klassi. Mitmel sellisel tüübil on olemas sümmeetriakeskpunkt: arvutüüpidel on selleks 0, suurusvahekorratüübil EQ. Seepärast võiks olla defineeritud ka meetod `kese`, mille väärtus sellistel tüüpidel oleks tüübi sümmeetriakeskpunkt.

Sümmeetriakeset ei ole siiski kõigil sümmeetrilistel tüüpidel: tõeväärtustest on raske üht või teist tüübi keskmeks pidada. Järelikult sobib sisse tuua eraldi tsentraalsümmeetriliste tüüpide klass, mis on sümmeetriliste tüüpide klassi alamklass ja mille esindajatunnuseks on meetod `kese`.

Selle jaoks kirjutame koodi

```
class (Sümm a) => TsentrSümm a where

  kese                                          .
    :: a
```
(221)

2. Alamklasside esindajate defineerimine. Esindajatüüpe klassile, mis on deklareeritud mingite klasside alamklassiks, saab valida ainult selliste tüüpide seast, mis neisse ülemklassidesse juba kuuluvad.

Klassideklaratsiooni (221) olemasolul saame tüübid Ordering ja Int viia ka tsentraalsümmeetriliste tüüpide klassi tuttaval kujul deklaratsioonidega

```
instance TsentrSümm Ordering where

  kese                                          ,
    = EQ
```
(222)

```
instance TsentrSümm Int where

  kese                                          .
    = 0
```
(223)

Interaktiivses keskkonnas on nüüd lihtne kontrollida, et tüübiannotatsiooniga avaldised `kese :: Ordering` ja `kese :: Int` annavad väärtustamisel tulemuseks vastavalt `EQ` ja `0`. Meetodi `kese` annoteerimine mõne muu tüübiga, näiteks tõeväärtusetüübiga, annab tüübivea.

Deklaratsioonid (222) ja (223) töötavad ainult juhul, kui Ordering ja Int on pandud kuuluma ka sümmeetriliste tüüpide klassi näiteks deklaratsioonidega (216) ja (215).

Kummatigi saab klassi meetodeid defineerida alamklassi meetodite kaudu.

Näiteks saab deklareerida tüübi Double sümmeetriliseks ja tsentraalsümmeetriliseks, määratledes vastandelemendi keskme suhtes sümmeetrilisena, st defineerides sümmeetriliste tüüpide klassi meetodi `vastand` tema tsentraalsümmeetriliste tüüpide alamklassi meetodi `kese` kaudu. Kese on loomulik võtta nulliks. Saame dek-

laratsioonid

```
instance Sümm Double where

  vastand x                    ,                    (224)
    = kese + (kese - x)

instance TsentrSümm Double where

  kese                                  ,           (225)
    = 0
```

mis koos töötavad laitmatult.

Nüüd kui millegipärast on soov saada muutuja `vastand` väärtuseks peegeldus mõne muu arvu, näiteks 1 suhtes, siis võib jätta deklaratsiooni (224) muutmata ja asendada esindajadeklaratsioon (225) koodiga

```
instance TsentrSümm Double where

  kese                                  .
    = 1
```

Alamklassi meetodeid saab kasutada küll esindajadeklaratsioonides, kui samad tüübid kuuluvad ka alamklassi, mitte aga vaikedefinitsioonides, sest alamklassi ei kuulu mitte iga tüüp klassis. Vaikedeklaratsioonides saab aga kasutada ülemklasside meetodeid — nende olemasolu on garanteeritud.

Olgu meil veel tahtmine defineerida klass selliste tüüpide jaoks, mida on võimalik ja realistlik programmi töö käigus ammendavalt läbi vaadata. Klassis peaks olema meetod `kõik`, mille väärtuseks oleks list, mille elemendid on parajasti kõik konkreetsesse tüüpi kuuluvad andmed.

Loomulik on nõue, et sellised tüübid peavad ühtlasi kuuluma klassi Enum. Kuid sellest ei piisa — Enum sisaldab ka lõpmatuid tüüpe nagu Integer ja isegi ujukomaarvutüübid. Nõuame lisaks, et meid huvitavad tüübid peavad kuuluma ka klassi Bounded, milles on olemas meetodid `minBound` ja `maxBound`, mis tähendavad vastavalt vähimat ja suurimat annet tüübis.

Võime kirjutada definitsiooni

```
class (Enum a, Bounded a) => Fin a where

  kõik
    :: [a]                                  .       (226)

  kõik
    = [minBound .. maxBound]
```

Selle tulemusel on uus klass korraga klasside Enum ja Bounded alamklass. Seega kõigil "lõplikel" tüüpidel on olemas `fromEnum`, `toEnum`, aritmeetilise jada süntaks ning ka `minBound`, `maxBound`. Praktiliselt saavutatakse see muidugi tänu nõudele, et esindajaks saab määrata vaid selliseid tüüpe, mis ülemklasside Enum ja Bounded esindajad juba on.

Vaikedefinitsioon näitab ühe võimaliku viisi kõigi andmete listi loomiseks. Pangem tähele, et selles on kasutust leidnud aritmeetilise jada süntaks, mis tuleb ülemklassist Enum, ning `minBound` ja `maxBound` ülemklassist Bounded.

Deklaratsiooni (226) olemasolul saame näiteks tõeväärtusetüübi ja suurusvahekorratüübi uue klassi esindajaks määrata lihtsalt ridadega

```
instance Fin Bool where,                    (227)
```

```
instance Fin Ordering where,                (228)
```

sest ainus meetod `kõik` on klassideklaratsioonis juba defineeritud.

Ülesandeid

423. Otsida Hugsi teegist üles klasside Ord, Num, Real, Integral, Fractional, Floating, RealFrac definitsioonid ja saada neist aru.

424. Kirjutada oma moodulisse deklaratsioonid (226), (227), (228) ja testida interaktiivses keskkonnas meetodit `kõik` nii tõeväärtus- kui suurusvahekorratüübi jaoks.

425. Defineerida tüüp Int lõplike tüüpide klassi esindajaks, määrates meetodi `kõik` väärtuseks listi, kus kõik 4-baidised täisarvud on järjestatud absoluutväärtuse järgi alates 0-st.

426. Defineerida klass, mis oleks nii tsentraalsümmeetriliste kui ka vahekaugustega tüüpide klassi alamklass ja kus oleksid defineeritud meetodid `kodeeri` ja `dekodeeri`, mis tähendaksid vastavalt kodeerimist täisarvuga tüübist Integer ja vastupidist operatsiooni. Anda neile meetoditele vaikedefinitsioonid, mis kasutavad ülemklasside meetodeid. Viia tüüp Int uue klassi esindajaks tühja kehaga esindajadeklaratsiooniga ja testida.

427. Defineerida klass nimega `Exp` arvutüüpide klassi alamklassina, millesse kuuluks meetod `aste`, mille kavandatav tähendus oleks tüüpi Double kuuluva arvu astendamine loodavasse klassi kuuluvat tüüpi arvuga. Kirjutada esindajadeklaratsioonid vähemalt ühe täisarvutüübi ja ühe ujukomaarvutüübi lisamiseks uude klassi, nii et meetodi `aste` definitsioon vastaks kavandatavale tähendusele.

6.1.3 Tingimuslikud esindajad

1. Tüübikontekst esindajadeklaratsioonis. Ka esindajadeklaratsiooni päisesse võib lisada tüübikonteksti. See seab piirangud tüübiparameetritele, millest sõltub see tüübipere, mille liikmed klassi esindajaks pannakse.

Näiteks võime ühe esindajadeklaratsiooniga kuulutada klassi Enum esindajaks kõik tüübid Maybe A, mille korral A ise kuulub klassi Enum. Täpsemalt, võime klassi Enum iga tüübi A jaoks antud selle klassi meetodid ühtsel viisil üle kanda vastavale tüübile Maybe A. Selle viib ellu deklaratsioon

```
instance (Enum a) => Enum (Maybe a) where

  fromEnum (Just x)
    = 1 + fromEnum x
  fromEnum _
    = 0

  toEnum n                                                    . (229)
    = case compare n 0 of
        GT
          -> Just (toEnum (n - 1))
        EQ
          -> Nothing
        _
          -> error "toEnum: negatiivne argument"
```

Klassis Enum on küll rohkem meetodeid, kuid ülejäänud meetodite jaoks on olemas vaikedefinitsioonid operaatorite `fromEnum` ja `toEnum` kaudu.

Pangem tähele, et meetodi `fromEnum` definitsioonis esineb `fromEnum` vasakul ja paremal erinevas tähenduses. See on tavaline üle laetava muutuja väärtuste varieerumine üle mitme tüübi. Kui vasakul on `fromEnum` väärtus tüüpi Maybe $A \to$ Int, siis paremas pooles tüüpi $A \to$ Int (`fromEnum` sellise variandi olemasolu garanteerib deklaratsiooni (229) tüübikontekst). Analoogiline on lugu muutuja `toEnum` esinemistega `toEnum` definitsioonis.

Deklaratsiooni (229) korral saab muuhulgas kasutada aritmeetilise jada süntaksit nurjumisega tüüpi elementidega listidel. Avaldise `[Just 'A' .. Just 'D']` väärtus näiteks on lõplik list elementidega Just A, Just B, Just C, Just D.

Samas pole deklaratsioon (229) mitte igas suhtes mõistlik. See, kui Nothing saab täisarvkoodi 0 ja ülejäänud andmed 1 võrra suurema arvu kui vastavad Just argumendid, sobib juhul, kui originaaltüüpi andmete täisarvkoodid on naturaalarvud; negatiivsete koodide korral, mida tuleb ka ette, varjutaks Nothing midagi ära.

Sarnaselt võime soovida ka näiteks laiendada oma sümmeetriliste tüüpide klassi üle listitüüpide, mille elemenditüübid on sümmeetrilised; antud listi vastandlist võiks olla selline, kus kõik elemendid on asendatud vastandelementidega ja nende järjekord on algsega võrreldes vastupidine. Siis sobib kood

```
instance (Sümm a) => Sümm [a] where

  vastand
    = foldl (\ as x -> vastand x : as) []
```
. (230)

Siingi esineb defineeritav meetod paremas pooles teist tüüpi väärtusega kui vasakus.

Tingimuslikud esindajadeklaratsioonid võimaldavad viia funktsioonitüüpe klassi Show sisukamalt kui kood (220), mis andis kõigile funktsioonidele sama sõnekuju.

Nimelt võiks sõnes ära tuua kõik võimalikud funktsiooni argumendid koos väärtusega. Selleks peab nii argumendi- kui väärtusetüüp kuuluma klassi Show. Et kõik võimalikud argumendid saaks läbi käia, on vaja koodiga (226) defineeritud klassi meetodit `kõik`, mis lisab tüübikonteksti veel ühe tingimuse.

Meid huvitab otseselt vaid meetod `show`, kuid pöörame tähelepanu teisele sama klassi meetodile `showsPrec`. See võimaldab sõneksteisendust kodeerida akumulaatoriga vältimaks keeruka ehitusega andmete osade sõnekujude kopeerimist.

Täpsemalt, muutuja `showsPrec` väärtus on funktsioon, mis võtab argumendiks (lühikese) täisarvu, sõneks teisendatava andme a ja mingi juba valmis sõne s ja annab tulemuseks a esituse sõnena, millele samas sõnes järgneb s. Esimene argument mängib rolli vaid juhul, kui sõnekujus tuleb märkida komponentide piire (nt sulgudega), tavaliselt võib seda argumenti ignoreerida (või anda muutmatult edasi).

Nii võiksime saada esindajadeklaratsiooni

```
instance (Show a, Show b, Fin a) => Show (a -> b) where

  showsPrec n f as
    = let
        lisa k
          = (++) ", " .
            showsPrec n k .
            (++) " -> " .
            showsPrec n (f k)
        _ : _ : s
          = foldr lisa ('}' : as) kõik
      in
      '{' : s
```
(231)

(kui `showsPrec` on defineeritud, siis `show` definitsiooni pole vaja anda, sest meetodi `show` jaoks on klassideklaratsioonis vaikedefinitsioon meetodi `showsPrec` kaudu). Funktsiooni sõneesituseks saab sellega argument-väärtus-paaride komadega eraldatud loend looksulgudes, kus argumenti ja väärtust eraldab noolemärk.

Lastes interaktiivses keskkonnas väärtustada näiteks operaatori `not`, saame nüüd vastuse "{False -> True, True -> False}".

Kui funktsioonitüüpi avaldis on üle laetud, siis tuleb sõneksteisendamisel annoteerida tüüp, muidu pole masinal selge, millise tüübi jaoks sõne koostada. Näiteks `fromEnum :: Bool -> Int` ja `fromEnum :: Ordering -> Int` interaktiivses keskkonnas töötavad, kuid paljas `fromEnum` annab ikka tüübivea.

Ülesandeid

428. Otsida Hugsi teegist üles deklaratsioon, mis defineerib listide võrduse, ja saada sellest aru.

429. Kirjutada esindajadeklaratsioon, mis laiendab klassi Num meetodid elementhaaval listitüüpidele tingimusel, et elemenditüüp kuulub klassi Num.

     Anda muutujale `fibs` uue klassi meetodite kaudu uus definitsioon koodi (150) eeskujul.

430. Milliseid kitsendusi rahuldavad funktsioonitüübid saab panna klassi Eq? Kirjutada esindajadeklaratsioon, mis seda teeb. Kõik meetodid peavad saama sisukalt defineerituks.

431. Milliseid kitsendusi rahuldavad funktsioonitüübid saab panna klassi Num, defineerides meetodid komponenthaaval? Kirjutada esindajadeklaratsioon, mis seda teeb. Kõik meetodid peavad saama sisukalt defineerituks.

432. Olgu antud signatuur ja definitsioon

```
intress
  :: Ordering -> Float'

intress LT
  = 1.5
intress EQ
  = 4.5
intress _
  = 12
```

     Mõte seisneb hoiuse kolme võimaliku aastaintressi korraga esitamises: argumendile LT vastab tavahoiuse intress, argumendile EQ tähtajalise hoiuse

intress ja argumendile GT investeerimisriskiga hoiuse oodatav intress. Kõik intressid on protsentides.

Ülesande 431 lahendusele toetudes defineerida muutuja `tulu` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja ujukomaarvu x ning annab tulemuseks x-krooniselt hoiuselt n aasta jooksul saadud tulu samas vormis (st funktsioonina tüüpi Ordering $\to$ Float, kus argumendile LT vastab tulu tavahoiuse intressi korral jne).

2. Rekursioon tüüpidel. Tingimused esindajadeklaratsiooni tüübikontekstis enamasti sisaldavad kitsendusi sama klassiga, millesse see deklaratsioon tüüpe lisab. Selline esindajadeklaratsioon on tsüklilise iseloomuga. Ta võib üksi anda klassikuuluvuse lõpmata paljudele järjest keerulisematele tüüpidele, sest esindajadeklaratsiooni tüübikontekstis nõutud klassikuuluvused võivad olla sellesama esindajadeklaratsiooniga tekitatud.

Esindajadeklaratsioon (229) viib tüübi Maybe A klassi Enum tingimusel, et A seal on. Ühtlasi viib sama deklaratsioon sel juhul klassi Enum ka tüübi Maybe (Maybe A), sest Maybe A tänu temale sinna saab, jne. Tekib lõpmatu jada

$$\text{Maybe } A, \text{Maybe (Maybe } A), \text{Maybe (Maybe (Maybe } A)) \ldots \qquad (232)$$

tüüpidest, mis kõik saavad klassi Enum tänu samale esindajadeklaratsioonile (229).

Samade omadustega on ka esindajadeklaratsioon (230). Tulemusena saavad sümmeetriliseks kõik listitüübid, mille elementitüüp juba on sümmeetriline, siis kõik listitüübid, mille elementitüübid on sellised listitüübid, jne lõpmatuseni.

Kui funktsioonitüüp $A \to B$ kuulub klassi Show ja mingi C on lõplik ja kuulub ka klassi Show, siis tüüp $C \to A \to B$ kuulub deklaratsiooni (231) järgi klassi Show. Ka seda üleminekut saab lõpmatu arv kordi jätkata. Järelikult saab edukalt sõneks teisendada ka avaldisi, mille väärtuseks on kuitahes suure argumentide arvuga karritatud funktsioonid.

Näiteks lastes interaktiivses keskkonnas väärtustada `(&&)`, saame vastuseks sisuliselt konjunktsiooni tõeväärtustabeli.

Meetodid sellises esindajadeklaratsioonis defineeritakse sisuliselt rekursiooniga tüübi ehituse järgi: üle laetava muutuja väärtus keerulisema ehitusega tüübis väljendatakse sama muutuja väärtuse kaudu lihtsama ehitusega tüübis. Mõeldes ilmutamata tüübiparameetrile, on see tõeline rekursioon.

Niisiis on meetodite definitsioonid koodis (229) omamoodi rekursiivsed, nagu ka esmapilk peale vaadates vihjab: `fromEnum` definitsiooni paremas pooles esineb `fromEnum` ja `toEnum` definitsiooni paremas pooles `toEnum`.

Deklaratsioon (229) annab meetoditele `fromEnum` ja `toEnum` sisu kõigi tüüpide jaoks jadas (232), kusjuures iga tüübi korral defineeritakse meetod ühe võrra väiksema Maybe rakenduste arvuga tüübil töötava sama meetodi variandi kaudu.

Ülesandeid

433. Luua ja lugeda interaktiivses keskkonnas sisse moodul, milles on esindajadeklaratsioonid (229) ja (230). Testida klassi Enum meetodeid tüüpidel kujuga Maybe (Maybe A) ja Maybe (Maybe (Maybe A)) ning sümmeetriliste tüüpide klassi meetodeid listidel, mille elemendid on omakorda listid.

434. Miks muutuja `compare` väärtustamine interaktiivses keskkonnas annab tüübivea ka esindajadeklaratsiooni (220) olemasolul? Sisestada interaktiivses keskkonnas selle muutujaga samaväärne avaldis, mille puhul viga ei teki.

435. Olgu antud signatuur

```
matKorruta
  :: (
       Fin a, Show a,
       Fin b, Show b,
       Fin c, Show c,
       Num p
     )
  => (a -> b -> p) -> (b -> c -> p) -> a -> c -> p
```

Funktsioone m tüübist $A \to B \to P$ interpreteerime maatriksitena, kus iga andme a jaoks tüübist A on üks rida, iga andme b jaoks tüübist B üks veerg ja $m\,a\,b$ on a-le vastava rea ja b-le vastava veeru ristumisel asuv arv.

Vastavuses selle signatuuri ja tõlgendusega ning toetudes ülesande 431 lahendusele anda `matKorruta` väärtuseks maatriksite korrutamise operatsioon.

6.1.4 Piirangud tüübiavaldisele esindajadeklaratsioonis

Haskelli standard seab tüübiavaldistele esindajadeklaratsioonides küllaltki ranged piirangud. Haskelli laiendustes on neid märksa vähem.

1. Lõpmatu tüübirekursiooni vältimine. Esindajadeklaratsiooni päises ei tohi tüübiperet tähistava tüübiavaldisena kasutada paljast tüübimuutujat.

Mõtet sellisel esindajadeklaratsioonil ju oleks. Kui näiteks on tahtmine korraga sümmeetriliseks kuulutada kõik arvutüübid (vastandarvu leidmine on ju igas ar-

vutüübis olemas), siis võiks sobida deklaratsioon

```
instance (Num a) => Sümm a where

  vastand
    = negate
```
. (233)

See deklaratsioon on aga keelatud.

Mõeldes rekursioonile tüübi struktuuri järgi, on keelu põhjused ka ilmsed. Rekursiooni terminites on tegu sammuga, kus argument ei lähe väiksemaks. Kui see oleks lubatud, siis saaks kergesti kirjutada koodi, mille tüübikorrektsuse analüüs jääks lõpmatult tööle. Tüübianalüüs kui staatilise analüüsi osa aga mingil juhul lõpmatult tööle jääda ei tohi.

Kui kood *à la* (233) oleks lubatud, siis võiks kirjutada juurde ka libadeklaratsiooni

```
instance (Sümm a) => Num a where.
```

Pärast seda võib näiteks sümbolitüübi korral jäädagi analüüsima, kas ta on sümmeetriline ja arvuline või mitte.

Tingimuslik esindajadeklaratsioon, kus tüübiperet tähistab muutuja, on idee poolest ühe klassi teiste alamklassiks kuulutamine; selle tarvis tuleks kasutada klassideklaratsioone. (Tüübikonteksti puudumisel defineeriks muutujast koosneva tüübiperega esindajadeklaratsioon universaalse, kõiki tüüpe sisaldava klassi, millel pole lihtsalt mõtet.)

Esindajadeklaratsioon (233) sisuliselt ütleks, et sümmeetriliste tüüpide klass on arvutüüpide klassi ülemklass. Antud olukorras kahjuks ei aita ka klassideklaratsioonid, sest Num on standardteegis juba defineeritud ja defineeritud klassidele saab lisada vaid alamklasse, mitte ülemklasse.

2. Mitmesuse vältimine. Teine olukord, mida Haskelli standardis on välditud, on sama tüübi sattumine samma klassi mitme esindajadeklaratsiooni kaudu.

Selle tõttu on nõutud, et tüübiavaldis esindajadeklaratsiooni päises peab olema kujul

$$\mathfrak{c}\ \mathfrak{x}_1\ \ldots\ \mathfrak{x}_l,$$

kus $l > 0$ ning $\mathfrak{c}$ on tüübikonstruktor (mitte sünonüüm) ja $\mathfrak{x}_1, \ldots, \mathfrak{x}_l$ paarikaupa erinevad muutujad.

Näiteks esindajadeklaratsioonides (215) ja (216) on $l = 0$ (argumente pole) ja vastavalt `c = Int` ja `c = Ordering`. Esindajadeklaratsioonis (229) on $l = 1$ ning `c = Maybe` ainsa argumendiga `a`. Deklaratsioonis (220), samuti deklaratsioonis (231), on $l = 2$ ning `c` funktsioonitüübikonstruktor.

See nõue välistab esindajadeklaratsiooni päise tüübiavaldise puhul korraga kolm asjaolu:

- argumendi positsioonis esineb muutujast keerukam alamavaldis;
- sama muutuja esineb mitmes positsioonis;
- tüübikonstruktori kohal on rakendatud tüübimuutujat (-sünonüümi).

Nende kitsenduste korral piisab süsteemil sama tüübi korduvalt samma klassi sattumise vältimiseks kontrollida, et tüübiavaldised erinevate esindajadeklaratsioonide päises oleksid erinevad. Ilma nende kitsendusteta saaksid erinevate avaldiste kirjeldatud pered sisaldada ühiseid tüüpe, mistõttu läheks vaja keerulisemat analüüsi.

Näiteks on keelatud esindajadeklaratsioonid päisega

```
instance Sümm [[a]] where
```

sest listikonstruktori argumendiks on omakorda listikonstruktoriga ehitatud tüübiavaldis, mitte muutuja. Koos võimaliku esindajadeklaratsiooniga (230) tekitaks see probleemi, kumma esindajadeklaratsiooni definitsioonid meetoditele listide listide tüübi jaoks ikkagi kehtivad.

Samuti on keelatud päis

```
instance Sümm (a , a) where
```

sest muutuja `a` esineb korduvalt. See tekitaks mitmesuse koos esindajadeklaratsiooniga, mille päises on üldisem tüübiavaldis `(a , b)`.

Tüübisünonüümi keeld välistab näiteks päise

```
instance Sümm String where
```

Ka see tekitaks koos deklaratsiooniga (230) mitmesuse.

6.2 Algebralised tüübid

Kui uute klasside tekitamine kõrvale jätta, oleme seni piirdunud Haskelli eeldefineeritud objektidega. Definitsioonid ei loonud uusi objekte, vaid kirjeldasid juba olemasolevaid ja sidusid nendega uusi muutujaid.

Programmeerija saab aga kirjeldada ka täiesti uusi tüüpe ja andmeid. Uute andmete sissetoomine nõuab seejuures alati ka uute tüüpide sissetoomist, pole võimalik näiteks uute andmete lisamine juba olemasolevasse tüüpi.

Uute objektide sissetoomine tähendab alati uute konstruktorite defineerimist: selleks, et uusi objekte vanadest eristada, tuleb neid koostada uute, endistest erinevate konstruktorite abil.

Vabadus uusi objekte konstrueerida võimaldab asju, mis reaalses elus millegi olulise poolest erinevad, mugavalt ka koodis erinevat tüüpi andmetena kujutada. Kood, mis on kirjutatud ülesande püstitusest tulenevate struktuuride terminites, on inimesele kergemini arusaadav. Samuti on temasse sisse tehtud vigadest tavalisest suurem osa tüübivead ja ilmnevad kompileerimise käigus, mitte täitmise ajal täitmisaegsete vigadena või, veel hullem, arvutuste tulemustes, kus neid kaua aega keegi ei aimagi olevat.

Tüüpe, mida saab kirjeldada Haskelli uute tüüpide defineerimise võimaluste piires, nimetatakse **algebralisteks**, arvatavasti on põhjuseks teoorias ilmnev analoogia sellega, kuidas esituvad üldises algebras nn algebralised arvud.

6.2.1 Tüüpide defineerimine

Enamasti sobib uute tüüpide ja neisse kuuluvate andmete sissetoomiseks deklaratsioon kujul

```
data t
  = a1  |  ...  |  an ,
```
(234)

kus `t` on uut tüüpi väljendav tüübiavaldis ning $\mathtt{a}_1, \ldots, \mathtt{a}_n$ on nn **alternatiivid**. Haskell 98 nõuab, et $n > 0$[1]. Deklaratsioon kujul (234) peab olema moodulis välistasemel, ta ei saa olla lokaalne.

1. Üksiku tüübi defineerimine. Vaatleme algul lihtsamat juhtu, kus defineeritakse korraga ainult üks tüüp. Siis kujutab `t` deklaratsioonis (234) endast lihtsalt uue tüübi nime. Kuna süntaktiliselt on tegemist tüübikonstruktoriga, mitte muutujaga, peab nimi algama suurtähega.

Iga alternatiiv deklaratsioonis (234) esindab uude tüüpi kuuluvate andmete üht võimalikku kuju. Koos kirjeldavad nad kõik sellesse tüüpi kuuluvad normaalsed andmed; neile lisandub veel $\perp$.

[1] Haskell 2010 lubab ka tühja paremat poolt ehk juhtu $n = 0$.

Iga andmete kuju eristab teistest unikaalne andmekonstruktor, mille rakendamine just selliseid andmeid tekitab. Konstruktori rakendamine n-ö vormistab andmestruktuuri, konstruktori argumendidest saavad tema väljad, analoogiliselt sellega, kuidas paarikonstruktor oma kaks argumenti üheks paariks kokku seob.

Alternatiiv deklaratsioonis (234) täpsemalt kujutab endast vastava andmekuju konstruktori täisargumenteeritud rakenduse šablooni, kus reaalsete argumentide kohal on nende tüüpe väljendavad avaldised. Seega on iga konstruktori argumentide tüübid fikseeritud. Sellest saavad ka konstruktorid ise omale kindlalt määratud tüübi. Konstruktoritele seetõttu tüübisignatuure andma ei pea ja Haskelli standard seda ka ei võimalda.

Andmekonstruktorite nimed peavad olema reeglipärased. Nende rakendus võib olla karritatud ja seejuures vastavalt soovile prefiksne või infiksne. Andmekonstruktori argumentide arv võib olla ka 0; see tähendab, et ühtegi rakendamist ei toimu, konstruktor ise esindab annet uuest tüübist.

1.1. Loetelutüübid. Kõige lihtsamal juhul pole ühelgi konstruktoril argumente. Siis on defineeritav tüüp loetelutüüp.

Näiteks deklaratsioon

```
data Kuu
  = Jaanuar
  | Veebruar
  | Märts
  | Aprill
  | Mai
  | Juuni                                    (235)
  | Juuli
  | August
  | September
  | Oktoober
  | November
  | Detsember
```

tekitab uue loetelutüübi nimega `Kuu` koos temasse kuuluva 12 andmega, mis on seotud uute konstruktoritega `Jaanuar`, `Veebruar`, `Märts`, `Aprill`, `Mai`, `Juuni`, `Juuli`, `August`, `September`, `Oktoober`, `November`, `Detsember`.

Kuutüüpi andmeid saab kasutada kalendrikuudega seotud arvutuste programmeerimiseks, kui on soov eristada kalendrikuid muudest andmetest ja mitte kasutada kuude tähistamiseks näiteks täisarve 1 kuni 12. Inimestena võtame andmeid koodiga (235) defineeritud tüübist kui kalendrikuid; masina jaoks on tegu lihtsalt mingite

andmetega, mis eristuvad kõigisse teistesse tüüpidesse kuuluvatest andmetest.

Eraldi kuutüübi kasutamine kuudega opereerimiseks teeb koodi tunduvalt arusaadavamaks võrreldes sellega, kui kuid tähistavad täisarvud 1 kuni 12. Arvud võivad sealsamas koodis tähistada ka mingeid muid reaalse elu suurusi ja sel juhul oleks koodi lugejale kuude ja teiste arvuliste suuruste eristamine vaevanõudev. Kui aga lugeja näeb koodis kuutüüpi andmekonstruktorit või mõnd muud kuutüüpi avaldist, siis see ütleb talle kohe, et mõeldud on teatavat kalendrikuud. Kuutüübi sihipärase kasutamise korral ei saa koodi sisse jääda viga, kus kalendrikuu kohal on mõne muu tähendusega objekt, sest selline viga tuleb välja tüübikontrolli käigus enne koodi jooksutamist. Arvude kasutamise puhul tekib ka oht, et kuu kohale satub arv, millele ei vastagi ühtki kuud, näiteks 13 või mõni negatiivne arv, ja seda viga võib olla raske avastada; kuutüüpi kasutades midagi sellist juhtuda ei saa.

Põhjused on sarnased sellega, miks võrdlemise puhul kasutatakse võrdlussuhete esitamiseks andmeid LT, EQ, GT loetelutüübist Ordering, kuigi saaks kasutada ka arve, samuti miks tõeväärtustel on oma tüüp jne.

Deklaratsiooni (234) olemasolul on võimalik uut tüüpi ja tema andmeid arvutamiseks põhimõtteliselt kohe kasutada. Uute andmekonstruktoritega saab moodustada nii avaldisi kui ka näidiseid vastavalt üldistele reeglitele.

Näiteks kood

```
suvekuud
  :: [Kuu]'

suvekuud
  = [Juuni, Juuli, August]
```
(236)

annab muutuja `suvekuud` väärtuseks suvekuude listi.

Probleeme võib tekitada asjaolu, et uus tüüp ei kuulu vaikimisi ühessegi klassi, mistõttu ei saa tema andmeid omavahel võrrelda ega sõnena esitada.

Et panna uusi tüüpe klassidesse kuuluma, võib muidugi kasutada esindajadeklaratsioone. Haskellis on aga olemas võimalik uute tüüpide sissetoomisel kasutada klassikuuluvuste automaattuletust. Selleks tuleb deklaratsiooni (234) lõppu lisada võtmesõna **`deriving`** ja loend klassidest, kuhu uus tüüp tahetakse panna. Automaattuletus ei tööta iga klassi jaoks, vaid ainult mõnede eeldefineeritud lihtsamate klasside jaoks.

Automaattuletus rahuldab meid muidugi vaid juhul, kui meil pole meetodite väärtuse osas erisoove. Näiteks sõnekujud tuletab süsteem automaatselt otse nimede järgi.

Lugedes interaktiivses keskkonnas sisse mooduli, kus on deklaratsioon (235), võime väärtustada näiteks konstruktori `Jaanuar`. Kuid oodatava vastuse "Jaanuar" asemel saame tüübivea, sest uus tüüp ei kuulu klassi Show.

Et kuutüüp läheks automaatselt klassi Show, piisab definitsiooni (235) lõppu lisada rida

```
deriving (Show).
```
(237)

Klassi Eq meetodi == väärtuseks paneb süsteem automaattuletusel predikaadi, mis normaalsetel andmetel töötab kui tegelik võrdus (aga kui mõni argument on $\perp$, on ka tulemus $\perp$). Klassi Ord meetodite tähenduse tuletab süsteem vastavalt alternatiivide järjekorrale tüübidefinitsioonis — eespool asuva kujuga andmed loetakse väiksemaks. See näitab, et automaattuletuse kasutamisel on alternatiivide järjekord definitsioonis oluline.

Funktsiooni, mis võtab argumendiks aasta ja kuu ning arvutab päevade arvu selle aasta selles kuus, saab esitada koodiga

```
päevi
  :: (Integer , Kuu) -> Int'

päevi (y , m)
  = case m of
      Aprill
        -> 30
      Juuni
        -> 30
      September
        -> 30
      November
        -> 30
      Veebruar                .
        | y `mod` 400 == 0
          -> 29
        | y `mod` 100 == 0
          -> 28
        | y `mod` 4 == 0
          -> 29
        | otherwise
          -> 28

      _
        -> 31
```
(238)

Definitsioon (238) on silmatorkavalt kohmakas. Paremas pooles on paljuhargnev valikuavaldis, mille nelja juhu parem pool on ühesugune. Programmeerijana me tahaks need neli juhtu kirjeldada ühekorraga, et vaeva kokku hoida.

Loomulik tee definitsiooni kohmakuse kaotamiseks oleks koondada ühesuguse päevade arvuga kuud listidesse ja kontrollida argumentkuu kuuluvust neisse. Tehes koodi (238) selle idee järgi ümber, saame definitsiooni

```
päevi (y , m)
  | elem m
    [
      Jaanuar, Märts, Mai, Juuli,
      August, Oktoober, Detsember
    ]
    = 31
  | elem m [Aprill, Juuni, September, November]
    = 30                                                    . (239)
  | y `mod` 400 == 0
    = 29
  | y `mod` 100 == 0
    = 28
  | y `mod` 4 == 0
    = 29
  | otherwise
    = 28
```

See kood aga niisama tööle ei hakka, sest ta nõuab kontrollimist, kas mingi kuu võrdub kuude listi mingi elemendiga — selleks peaks võrdus olema kuutüübil defineeritud ehk kuutüüp kuuluma klassi Eq. Aitab, kui rea (237) asemel täiendada definitsiooni (235) reaga

```
    deriving (Show, Eq).
```

Kuutüübi klassi Ord kuuluvuse automaatsel tuletamisel tekiks võrdlusoperatsioon, mille järgi aastas eespool paiknevad kuud on tagapool paiknevatest väiksemad; see tuleneb otse kuude järjestusest kuutüübi definitsioonis.

Loetelutüüpide puhul töötab ka klassi Enum kuuluvuse automaattuletus. See annab loetelutüüpi andmetele lihtsa ligipääsu järjekorranumbrite kaudu. Esimene konstruktor saab järjekorranumbri 0, järgmine 1 jne.

Seega kuutüübi korral ollakse traditsioonilise numeratsiooni suhtes 1 võrra nihkes.

Edaspidi on oluline kuutüübi kuuluvus kõigisse seni mainitud klassidesse, nii et eeldame, et definitsiooni (235) on täiendatud reaga

```
deriving (Show, Eq, Ord, Enum).
```

Tänu kuulumisele klassi Enum saab muutuja `päevi` definitsiooni (239) veelgi lihtsustada, kirjutades

```
päevi (y , m)
  | elem jrk [1, 3, 5, 7, 8, 10, 12]
    = 31
  | elem jrk [4, 6, 9, 11]
    = 30
  | y `mod` 400 == 0
    = 29
  | y `mod` 100 == 0
    = 28                                    .
  | y `mod` 4 == 0
    = 29
  | otherwise
    = 28
  where
    jrk
      = fromEnum m + 1
```

(240)

Koodiga (236) defineeritud muutuja `suvekuud` samaväärne definitsioon aga oleks

```
suvekuud
  = [Juuni .. August].
```

Klassi Enum kuuluvate tüüpide jaoks on olemas operaatorid `succ` ja `pred`, mis tähendavad vastavalt järgmise ja eelmise andme võtmist. Neist esimese abil saame kirjeldada näiteks funktsiooni, mis leiab etteantud aasta kuule järgneva kuu, koodiga

```
järgmKuu
  :: (Integer , Kuu) -> (Integer , Kuu)'

järgmKuu (y , m)
  = case m of
      Detsember
        -> (succ y , Jaanuar) .
      _
        -> (y , succ m)
```

(241)

Ülesandeid

436. Otsida Hugsi teegist üles tüüpide Bool ja Ordering definitsioonid ja saada neist aru.

437. Kirjutada definitsioon (235) oma moodulisse ja viia kuutüüp klassi Enum. Defineerida muutuja `tagurpidiAasta` väärtuseks list, milles on parajasti kõik kalendrikuud aastas esinemisele vastupidises järjekorras.

438. Defineerida muutuja `eelmKuu` väärtuseks funktsioon, mis leiab antud aasta antud kuule eelneva kuu. Tüüp peab olema sama mis koodiga (241) defineeritud muutujal `järgmKuu`.

439. Defineerida loetelutüüp `Nädalapäev`, millesse kuuluvad andmed vastavad nädalapäevadele. Seejärel defineerida muutuja `tööpäev` väärtuseks predikaat, mis võtab argumendiks nädalapäeva ja annab True parajasti juhul, kui argument on tööpäev.

440. Defineerida loetelutüüp `Viker`, millesse kuuluvad andmed vastavad värvidele värviringis. Defineerida muutuja `järgmVärv` väärtuseks funktsioon, mis võtab argumendiks värvi ja annab tulemuseks ringis järgmise värvi. Analoogselt defineerida ka muutuja `eelmVärv` ringis eelmise värvi leidmiseks.

1.2. Ühe konstruktoriga tüübid. Teine lihtne juht deklaratsioonist kujul (234) on selline, kus on ainult üks alternatiiv. Siis kõik normaalsed andmed uuest tüübist konstrueeritakse sama konstruktoriga. Tuntutest kuuluvad selliste tüüpide hulka näiteks paaritüüp, kus kõik normaalsed andmed on saadavad paarikonstruktoriga.

Tavaliselt pannakse ühe alternatiiviga tüüpide puhul andmekonstruktorile ja tüübikonstruktorile sama nimi, aga muidugi ei ole see kohustuslik. See, et tüübi nimi ja andmekonstruktori nimi langevad kokku, ei sega kuidagi, sest konteksti järgi on alati selge, kas nime konkreetne esinemine peab tähendama tüüpi või annet.

Näiteks võib kuutüübi kõrval tuua sisse kuupäevatüübi deklaratsiooniga

```
data Daatum
  = Daatum Integer Kuu Int.                    (242)
  deriving (Show, Eq, Ord)
```

Kuupäevatüübi nimeks on valitud `Daatum`, see nähtub deklaratsiooni (242) võrduse vasakust poolest. Parem pool ütleb, et iga anne selles tüübis koostatakse andmekonstruktoriga, mille nimi on samuti `Daatum` ja mis võtab kolm argumenti:

täisarvu, kalendrikuu ja veel ühe täisarvu. Lepime kokku, et esimese täisarvu mõte on mängida aastaarvu rolli ja viimase täisarvu mõte on mängida päeva numbri rolli.

Niisiis on nimi `Daatum` deklaratsiooni (242) erinevates pooltes kasutatud erinevas tähenduses: vasakul on ta tüüp ehk 0 argumendiga tüübikonstruktor, paremal aga 3 argumendiga andmekonstruktor. Andmekonstruktor `Daatum` on tüüpi `Integer -> Kuu -> Int -> Daatum` (selles tüübiavaldises tähendab `Daatum` tüüpi).

Näiteks oleks Tartu rahu sõlmimise päev deklaratsiooniga (242) sisse toodud tüüpi avaldisena üles kirjutatav kujul `Daatum 1920 Veebruar 2`.

Kasutades ka koodiga (238) või (239) või (240) defineeritud operaatorit `päevi` ja koodiga (241) defineeritud operaatorit `järgmKuu`, saab nüüd funktsiooni, mis leiab etteantud kuupäevale järgneva kuupäeva, kirjeldada koodiga

```
järgmPäev
  :: Daatum -> Daatum'

järgmPäev (Daatum y m d)
  | päevi (y , m) > d
    = Daatum y m (d + 1)
  | otherwise
    = let                          .
        (y' , m')
          = järgmKuu (y , m)
      in
      Daatum y' m' 1
```
(243)

Jällegi on omadefineeritud konstruktorit kasutatud argumendinäidises. See näidis sobitub iga normaalse kuupäevatüüpi andmega ja sobitamise tulemusel saab `y` väärtuseks kuupäeva esimese välja, milleks on aastat tähistav arv, `m` saab väärtuseks teise välja ehk kuu ning `d` saab väärtuseks päeva numbri.

Päevatüübil on siiski juures samad vead, mille vastu kuutüübi sissetoomisega tegutsesime. Kõrgema nõudlikkuse korral võiksime ka aasta- ja päevanumbrite tähistamiseks tuua sisse spetsiaalsed tüübid, nagu tegime kuudega, näiteks deklareerides

```
data Aasta
  = Aasta Integer'
```
(244)

```
data Päev
  = Päev Int
```
(245)

ja kasutades definitsioonis (242) neid. Tüüpidesse `Aasta` ja `Päev` kuuluvad andmed on ühe täisarvulise väljaga andmestruktuurid. Siis ei ole võimalik kuupäevi ega aastaid koodis muude arvuliste andmetega segi ajada, sest tüübikontroll avastaks selle.

Kuupäevadega on asi siiski hullem kui kuudega, sest ka definitsioonide (244), (245) kontekstis saab kirjutada olematuid kuupäevi, kus päeva number on negatiivne või suurem kui antud kuus päevi. Seda olukorda on aga juba tülikas parandada, sest erinevates kuudes on päevade arv erinev.

Kasutame järgnevas deklaratsiooniga (242) antud kuupäevatüüpi edasi. Viidatud puuduse kompenseerimiseks kirjeldame koodiga

```
olemas
  :: Daatum -> Bool'

olemas (Daatum y m d)
  = 0 < d && d <= päevi (y , m)
```
(246)

predikaadi, mis kontrollib, kas etteantud kuupäevatüüpi anne väljendab reaalset kuupäeva. Päeva legaalsust tuleks kontrollida enne selliste operaatorite nagu `järgmPäev` väljakutset, muidu võime saada ootamatuid tulemusi.

Automaattuletusega saadav väljadega andmestruktuuride sõneksteisendamine esitab andmestruktuurid selliselt, kuidas Haskellis saaks neid korrektselt üles kirjutada. Võrduse automaattuletusel loetakse võrdseteks parajasti need sama konstruktoriga ehitatud struktuurid, mille kõik väljad on vastavalt võrdsed väljatüübi jaoks defineeritud operaatori == mõttes.

Automaattuletusel saadav järjestus on olemuselt leksikograafiline. Struktuure loetakse kui sõnu, kus tähtedeks on struktuuri väljad. Kui struktuurid on koostatud sama konstruktoriga, siis on "sõnad" ühepikkused ja vastavad "tähed" on sama tüüpi. Sellisel juhul otsustab esimene erinevus, kumb sõna on teisest eespool, kusjuures iga välja hindamine toimub vastavalt klassi Ord meetodite väärtusele selle välja tüübi jaoks.

Pangem tähele, et nende eeskirjade järgi klassikuuluvuse kehtestamiseks on tarvilik, et iga välja tüüp samma klassi juba kuuluks. Kui see nii pole, siis püüd automaattuletust kasutada annab tüübivea.

Automaattuletusel saadav võrdus kuupäevatüübil vastab päevade reaalsele võrdusele, sest üht ja sama päeva esitab ainult üks kuupäevatüüpi anne.

Kui tegelda ainult legaalsete kuupäevadega, siis vastab ka automaattuletusel saadav järjestus päevade ajalisele järgnevusele. Siin mängib otsustavat rolli väljade järjestus kuupäevatüübi definitsioonis.

Näiteks predikaadi, mis antud kuupäeva kohta kontrollib, kas see kuulub Eesti esimesse omariiklusaega (eeldame, et kuupäeva legaalsust pole kontrollida vaja), võik-

sime kirjeldada koodiga

```
omariiklus1
  :: Daatum -> Bool'

omariiklus1 d
  = Daatum 1918 Veebruar 24 < d &&  .
    d < Daatum 1940 August 6
```
(247)

Ülesandeid

441. Defineerida muutuja `sünnipäevad` väärtuseks funktsioon, mis võtab argumendiks kuupäeva d ja annab tulemuseks lõpmatu listi kuupäevadest, mil tähistatakse päeval d sündinud inimese sünnipäeva — eeldame, et 29. veebruaril sündinud inimese sünnipäeva tähistatakse 29. veebruaril, kui see kuupäev antud aastas on, muidu 28. veebruaril.

442. Defineerida muutuja `eelmPäev` väärtuseks funktsioon, mis võtab argumendiks kuupäeva ja annab tulemuseks talle vahetult eelneva kuupäeva.

443. Defineerida muutuja `vaheAastates` väärtuseks funktsioon, mis võtab argumendiks kuupäevad d_1, d_2 ja annab väärtuseks arvu, mitu täisaastat on kuupäeval d_2 möödas kuupäevast d_1.

444. Defineerida muutuja `päevaNr` väärtuseks funktsioon, mis võtab argumendiks kuupäeva ja annab tulemuseks sellele kuupäevale eelnevate päevade arvu samas aastas.

445. Kasutades ülesande 444 lahendust, defineerida muutuja `vahePäevades` väärtuseks funktsioon, mis võtab argumendiks kuupäevad d_1, d_2 ja annab väärtuseks päevade arvu kuupäevade d_1 ja d_2 vahel.

446. Automaatse tüübituletuse asemel defineerida kuupäevatüüp klassi Show esindajaks nii, et kuupäevade sõnekuju vastaks ISO standardile (st vormile "YYYY-MM-DD").

447. Defineerida aasta- ja päevanumbritüübid deklaratsioonidega (244), (245) ja teha kuupäevatüübi definitsioon (242) ümber nii, et uued tüübid oleksid sobivalt kasutatud. Muuta definitsioone (240) ja (241) nii, et nad sobivalt kasutaksid aastanumbritüüpi. Kohandada definitsioonid (243), (246), (247) muutunud oludele.

448. Kasutades ülesandes 439 defineeritud nädalapäevatüüpi, defineerida muutuja `nädalapäev` väärtuseks funktsioon, mis võtab argumendiks kuupäeva ja annab tulemuseks nädalapäeva, millele see kuupäev satub.

449. Nädalakalendris jaotatakse päevad mitte kuudesse, vaid nädalatesse; nädalad on samad nagu tavalises kalendris. Päevad identifitseeritakse aastaarvu, nädala numbri ja nädalapäeva kaudu. Nädal kuulub sellise numbriga aastasse, millise numbriga aastasse kuulub Gregooriuse kalendris tema neljapäev. Ühe aasta piires tähistatakse nädalaid järjestikuste täisarvudega alates 1-st.

    Kasutades ülesande 448 lahendust, defineerida nädalakalendripäevatüüp. Defineerida muutuja `olemas` väärtuseks predikaat, mis võtab argumendiks sellist tüüpi andme ja kontrollib, kas see päev on nädalakalendris olemas.

1.3. Mitme konstruktoriga tüübid. Mitut alternatiivi sisaldava definitsiooniga kujul (234) saab tekitada mitmel eri moel ehitatud struktuure hõlmavaid tüüpe. Tuntud tüüpidest on sellised summatüübid, mille andmed ehitatakse kas konstruktoriga `Left` või `Right` ja mõlemad võtavad ühe argumendi.

Näitena tüübidefinitsioonist, mis sisaldab mitu alternatiivi, mille konstruktoritel on argumendid, anname deklaratsiooni

```
data Kalendripäev
  = Gregooriuse Daatum
  | Juuliuse Daatum
  deriving (Show)
```
. (248)

Andmed selles tüübis väljendavad kuupäevi konkreetses kalendris — kas gregooriuse või juuliuse ehk, nagu öeldakse, vastavalt uues või vanas kalendris. Sellist tüüpi võib vaja minna juhul, kui soovitakse päevi talletada sel päeval kehtinud kalendri järgi. Definitsioon (248) on mõttekas tänu sellele, et nii juuliuse kui gregooriuse kalendris kasutatakse sama kuupäevakuju, mis on kodeeritud tüübis `Daatum`.

Seda tüüpi avaldisena kirjutatult oleks Tartu rahu sõlmimise päev kujul

```
Gregooriuse (Daatum 1920 Veebruar 2).
```

Kuigi Haskell ei luba olemasolevaid tüüpe laiendada ega kasutada tüübina olemasolevate tüüpide ühendit, siis mitme alternatiivse konstruktoriga ehitatavate andmestruktuuride tüübid võimaldavad seda kõike lavastada.

Tüüp `Kalendripäev` on sisuliselt kuupäevatüübi ühend oma koopiaga.

Ülesandeid

450. Defineerida muutuja `sordi` väärtuseks funktsioon, mis võtab argumendiks koodiga (248) defineeritud kalendripäevatüüpi andmete listi ja annab tulemuseks samade kalendripäevade listi, kus gregooriuse kalendri kuupäevad on ees

ja juuliuse kalendri kuupäevad taga. Sama kalendri kuupäevade omavaheline järjestus peab olema sama mis originaallistis.

451. Muuta koodiga (248) defineeritud kalendripäevatüüp klasside Eq ja Ord esindajaks selliselt, et sama päeva esitused erinevates kalendrites loetaks võrdseks ning järjestus vastaks ajalisele.

1.4. Keerulisemad tüübid. Deklaratsiooni (234) alternatiivides võib esineda kuitahes keerulisi tüübiavaldisi.

Olgu meil näiteks vaja tegelda olukordadega, kus kuupäev ei pruugi olla täpselt teada, kuid meil võib olla tema kohta teatavat informatsiooni. Siis võib olla sobiv kasutada definitsiooni

```
data EbaDaatum
  = Teadmata
  | Piirid Daatum Daatum
  | Valik [Daatum]
```
(249)

See deklaratsioon määratleb tüübi, mille elemendid on kolme sorti. Esimene variant on `Teadmata`, millel välju pole ja mis näitab, et meil puudub kuupäeva kohta igasugune informatsioon. Teiseks on konstruktoriga `Piirid` kahest kuupäevatüüpi väljast kokku pandud struktuurid, kus kaht kuupäeva võib mõista kui alumist ja ülemist ajalist tõket. Kolmandaks on konstruktoriga `Valik` ühest kuupäevade listi tüüpi väljast ehitatud andmed, kus listis on kõik võimalikud kuupäevad ükshaaval üles loetud. Sellist laadi annet esitab näiteks avaldis `Valik [Daatum 1918 Veebruar 24, Daatum 1991 August 20]`.

Tüüp `EbaDaatum` on ehituse poolest kuupäevade listi tüübi ja kuupäevade paari tüübi ühend, millesse on lisatud veel üks anne.

Kirjeldame kasutusnäitena predikaadi, mis võtab argumendiks ebamäärase kuupäevainfo p ja täpse kuupäeva d ning kontrollib, kas osaline info p lubab kuupäeva d; signatuur oleks

```
lubab
  :: EbaDaatum -> Daatum -> Bool
```

Eeldades, et olematuid kuupäevi ei kasutata, sobib kood

```
lubab ed d
  = case ed of
      Valik ds
        -> elem d ds
      Piirid a b
        -> a <= d && d <= b
      _
        -> True
```
(250)

Ülesandeid

452. Defineerida muutuja `valikuna` väärtuseks funktsioon, mis võtab argumendiks ebamäärase kuupäeva ja annab tulemuseks sama päevade hulka kirjeldava sama tüüpi andme, mis on esitatud valikuna kuupäevade listist.

453. Defineerida uus tüüp `Üksliige`, millesse kuuluvate andmetega saaks esitada astme ja eksponendi kujul üksliikmeid, st kujul x^a või a^x mingi arvu a ja muutuja x korral. Konstandid olgu tüüpi Double, muutujad sõnetüüpi.

2. Tüübiperede defineerimine. Et deklaratsiooniga (234) defineerida korraga terveid tüüpide peresid, peab deklaratsiooni vasakus pooles olema uus tüübikonstruktor koos argumendinäidistega. Haskelli standard lubab tüübinäidistena kasutada vaid muutujaid (seda nägime juba tüübisünonüümide defineerimisel). Parem pool on ehituselt selline nagu varem kirjeldatud, ainult et ta võib vasakus pooles sisse toodud muutujaid sisaldada.

Vasaku poole muutujate iga väärtustuse jaoks saame ühe uue tüübi. Täpsemalt, uue tüübikonstruktori tähenduseks on tüübifunktsioon, mis võtab vasaku poole iga argumendinäidise kohta ühe tüübiargumendi, ja annab neil tulemuseks ühe uue tüübi. Rohkem kui ühe argumendinäidise korral on tüübifunktsioon karritatud. Tulemustüüp koosneb andmetest, mis on kirjeldatud parema poolega samamoodi nagu varem, kusjuures lokaalsete muutujate väärtuseks on vastavad argumendid.

Suures osas on põhimõte sama nagu funktsionaalse vasaku poolega deklaratsioonide ja tüübisünonüümide deklaratsioonide puhul. Erinevalt neist on uue tüübi definitsiooni vasak pool alati täisargumenteeritud, sest parem pool esitab andmete kogumit, mitte kunagi tüübifunktsiooni.

Enamasti ongi uusi tüüpe mõistlik sisse tuua perede kaupa, et võimaldada polümorfsust. Kui definitsiooni (234) vasak pool sisaldab muutujaid, on alternatiive tähistavad andmekonstruktorid automaatselt polümorfsed, nad on kasutatavad tüübimuutujate kõigi väärtuste korral. Konstruktorite polümorfsusega oleme varasemast tuttavad, sest samamoodi polümorfsed on näiteks listikonstruktorid `[]` ja `:` ning nurjumisega tüüpi andmete konstruktorid `Nothing` ja `Just`.

Näiteks võib deklaratsioonis (249) kirjeldatud tüübiga `EbaDaatum` sarnase põhimõtte alusel koostatud tüüpi vaja minna peale kuupäevade ka teistlaadi andmete järgnevusvahemike ja valikute esitamiseks. Seepärast on mõistlik abstraheerida

kuupäevatüüp definitsioonis (249) välja suvaliseks tüübiks. Saame deklaratsiooni

```
data Eba d
  = Teadmata
  | Piirid d d'
  | Valik [d]
```
(251)

mis defineerib ühe parameetriga tüübipere. Konstruktori `Teadmata` tüübiks saab `Eba d`, konstruktori `Piirid` tüübiks `d -> d -> Eba d` ning konstruktori `Valik` tüübiks `[d] -> Eba d`.

Deklaratsiooni (251) olemasolul on korrektne näiteks avaldis `Piirid 0 5` tüübiga `Eba Integer` või üldiselt `(Num d) => Eba d`. Samuti on korrektsed avaldis `Piirid (-3.5) 8.9` üldise tüübiga `(Floating d) => Eba d`, avaldis `Valik "Aa"` tüübiga `Eba Char` jne.

Tüüp `Eba Daatum` on aga sisuliselt sama mis endine `EbaDaatum`. Muidugi ei saa tüüpe `Eba Daatum` ja `EbaDaatum` kasutada samas moodulis. Definitsioonid (249) ja (251) ei saa korraga jõus olla, sest defineerivad samad andmekonstruktorid.

Kui nüüd kirjeldame muutuja `onTeadmata` koodiga

```
onTeadmata Teadmata
  = True
onTeadmata _        ,
  = False
```
(252)

siis see muutuja on polümorfne, kasutatav andmete jaoks tüüpidest `Eba d` tüübimuutuja `d` suvalise väärtuse korral. Signatuuriks deklaratsiooni (252) juurde sobib

```
onTeadmata
  :: Eba d -> Bool.
```

Muutuja `lubab` definitsioon koodist (250) on aga ilma muutusteta interpreteeritav polümorfselt, tuleb vaid tüübisignatuur ära jätta või asendada signatuuriga

```
lubab
  :: (Ord d)          .
  => Eba d -> d -> Bool
```

Ülesandeid

454. Otsida Hugsi teegist tüübiperede Maybe ja Either definitsioonid ja saada neist aru.

455. Polümorfselt defineerida muutuja `võimalik` väärtuseks predikaat, mis kontrollib deklaratsiooniga (251) defineeritud tüüpi andme kohta, kas tema esitatav piirang on täidetav. Tüübiparameetri võib kitsendada klassiga Ord.

456. Defineerida kahe parameetriga tüübipere, mille iga konkreetne tüüp sisaldab parameetritega määratud tüüpidesse kuuluvate andmete paarid ja esimese parameetriga määratud tüüpi andmed üksikuna. Uute konstruktorite nimed valida vastavalt tähendusele.

    Defineerida muutuja `paarina` väärtuseks funktsioon, mis võtab sellist tüüpi andme argumendiks, üksikelemendid teisendab dubleerimise teel paarikujule ja paarid jätab puutumata.

457. Defineerida ülesandes 453 defineeritud tüüp ümber polümorfseks, kus konstanditüüp ja muutujatüüp võiksid olla suvalised.

458. Ülesandes 457 kirjutatud tüübidefinitsiooni kontekstis defineerida muutuja `isExp` väärtuseks funktsioon, mis annab True, kui argument on eksponendikujul, ja False, kui argument on astmekujul.

459. Ülesandes 457 kirjutatud tüübidefinitsiooni kontekstis defineerida muutuja `dominokuju` väärtuseks predikaat, mis võtab argumendiks üksliikmete listi ja annab tulemuseks True, kui list on lõplik ja igast kahest järjestikusest liikmest esimese astendaja võrdub teise astendatavaga, ja False, kui mainitud järjestikuste liikmete tingimus on mingis kohas rikutud.

6.2.2 Rekursiivselt defineeritud tüübid

Väga tihti läheb vaja andmestruktuure, mille ehitus on tsükliline selles mõttes, et tema mingi alamstruktuur on tervikuga sama tüüpi. Tüüpilised näited on mitmesugused kahendpuud ja muud puulaadsed struktuurid.

Sellise andmestruktuuri konstruktoril peab vähemalt ühe argumendi väärtus sisaldama endas konstrueeritava struktuuriga sama tüüpi struktuuri. Näiteks kahendpuudel on vasak ja parem haru omakorda kahendpuud.

Kui tahame sellise tüübi definitsiooni kirja panna kujul (234), peame paremas pooles konstruktori argumendi kohal kasutama defineeritavat tüüpi ennast. Seega on tegemist rekursiivse definitsiooniga. Arvestades funktsionaalsete keelte rekursioonilembust, on see muidugi lubatud, kuid tasub märkida, et siinkohal on rekursioon ainuvõimalik tee isegi tavapärastes imperatiivsetes keeltes.

Haskellis saavad rekursiivsed olla küll uue tüübi definitsioonid, kuid mitte tüübisünonüümidefinitsioonid. Kaudse rekursiooni puhul tähendab see, et rekursioon peab käima läbi vähemalt ühe uue tüübi definitsiooni.

1. Listilaadsed andmestruktuurid. Rekursiivse ehitusega on tuttavatest andmestruktuuridest listid: iga mittetühja listi saba (konstruktori `:` teine argument) on omakorda sama tüüpi list. Listid on küll Haskelli sisse ehitatud, nii et nende reaalset definitsiooni teegist ei leia, kuid lihtsasti on võimalik defineerida uusi tüüpe, mis sisaldavad sarnaseid andmestruktuure, kus elemendid paiknevad ühes jorus.

Lähtudes deklaratsiooni (234) mõttest ja listide ehitusest vastab listitüüpide perele tinglik definitsioon

```
data [a]
  = []
  | a : [a]
```
.

Interpreteerides seda pseudodefinitsiooni vastavalt *data*-deklaratsioonide mõttele, tõdeme, et kirjeldatav tüübipere sõltub ühest tüübiparameetrist, kusjuures tüübile A vastav tüüp selles peres sisaldab mingi andme, mida väljendab konstruktor `[]`, ning lisaks kõik sellised andmed, mis ehitatakse konstruktoriga `:` ühest A-tüüpi andmest ja kirjeldatavaga sama tüüpi struktuurist. Esimene alternatiiv on n-ö baasjuht, tuues sisse andme, millest alates kõik lõplikud listid on üles ehitatud, teine esitab n-ö rekursiooni sammu.

Defineerime sarnase, kuid juba legaalse näitena tüübid, millesse kuuluvad struktuurid on oma ehituselt samuti elementide järjendid nagu listid, kuid mille hulgas puudub tühi. Alternatiivideks oleksid baasjuhuna ühekomponendiline struktuur ja sammuna, nagu tavalistel listidel, komponendi lisamine sama tüüpi struktuuri. Teise alternatiivi ehitus peab vastavalt olema sama mis listitüüpide definitsioonil, kuid esimeses alternatiivis peab konstruktoril olema üks argument — üheelemendilise struktuuri ainus komponent. Saame definitsiooni

```
data Joru a
  = Üks a
  | Mitu a (Joru a)
  deriving (Show)
```
. (253)

Kui paneksime esimeseks alternatiiviks argumendita konstruktori, saaksime listitüüpidega struktuurilt samaväärsed tüübid: esimese alternatiivi konstruktor mängiks tühja listi rolli ja konstruktor `Mitu` kooloni rolli.

Definitsiooni (253) puhul aga on võimalikud avaldised tüübist `Joru Int` näiteks `Üks 0`, `Üks 1`, `Mitu 1 (Üks 2)`, `Mitu 3 (Mitu 5 (Üks 15))` jne, mis kõik esitavad struktuuri, kus vähemalt üks element.

Kui tüüp sisaldab andmeid, mille pärisosana esinevad sama tüüpi andmed, tuleb sellel tüübil töötavate funktsioonide kirjeldamisel loomuldasa kasutada rekursiooni.

Nii on see listide puhul ja samuti äsjadefineeritud tüübi puhul. Näiteks funktsiooni, mis leiab kogu jorustruktuuri elementide arvu, defineerib muutuja `joruPikkus` väärtuseks rekursiivne kood

```
joruPikkus (Mitu _ xj)
  = 1 + joruPikkus xj
joruPikkus _
  = 1
```

elementide summa arvutamise aga võib programmeerida koodiga

```
joruSumma (Mitu x xj)
  = x + joruSumma xj
joruSumma (Üks x)
  = x
```

Signatuurid on vastavalt

```
joruPikkus
  :: Joru a -> Int

joruSumma
  :: (Num a)
  => Joru a -> a
```

On võimalik defineerida ka listilaadsete struktuuride tüüp, millesse kuuluksid vaid lõpmatud ja osalised struktuurid (lõplikke mitte). Selleks piisab baasjuhu alternatiiv välja jätta. Valides elemendi lisamise konstruktoriks `:.`, sobib definitsioon

```
data LõpmatuList a
  = a :. LõpmatuList a
  deriving (Show)
```
(254)

Ülesandeid

460. Defineerida muutuja `joruElem` väärtuseks funktsioon, mis võtab argumendiks suvalise andme ja sama tüüpi andmete joru ja annab tulemuseks tõeväärtuse vastavalt sellele, kas see anne esineb jorus elemendina või mitte.

461. Defineerida muutuja `joruListiks` väärtuseks funktsioon, mis võtab argumendiks joru ja annab välja listi samade elementidega.

462. Defineerida muutuja `listJoruks` väärtuseks funktsioon, mis võtab argumendiks listi ja kui see on mittetühi, siis annab tulemuseks joru samade andmetega. Tühja listi korral lõpetagu arvutus vastava veateatega. Seda operaatorit kasutades kirjutada avaldis, mille väärtus on lõpmatu joru.

463. Testida koodiga (254) sisse toodud tüüpi lõpmatuid liste interaktiivse keskkonna käsurealt.

2. Puulaadsed andmestruktuurid. Algoritmikas nimetatakse puuks andmestruktuuri, kus andmed paiknevad nn tippudes, mis on organiseeritud hierarhiliselt, kusjuures kõik tipud selles hierarhias alluvad kas otse või kaudselt lõppkokkuvõttes üheleainsale tipule, mida nimetatakse juureks. Tippu, millele ei allu ühtki teist tippu, nimetatakse leheks.

Puust ja tippudest võiksime rääkida ka listilaadsete struktuuride juures, nt list on puu erijuht, kus iga tipp peale esimese allub vahetult ainult ühele tipule (eelmisele; esimene ei allu ühelegi) ja igas tipus paikneb üks element. Tipp on ometi abstraktsem mõiste kui element, sest mõnes struktuuris lubatakse samma tippu paigutada suvalisel arvul struktuuri elemente.

Puude kogumit nimetatakse metsaks. Tavaliselt on puud metsas nummerdatud või muidu kindlate nimede järgi identifitseeritavad.

2.1. Kahendpuud. Kahendpuus allub igale tipule maksimaalselt kaks tippu. Täpse määratluse järgi kahendpuu kas on tühi (ei sisalda ühtki tippu) või tema kõik tipud peale juure moodustavad omakorda kaks kahendpuud, mida nimetatakse vasakuks ja paremaks haruks. Hierarhia saame, lugedes puu harude juured puu juure alluvaiks.

See määratlus näitab vaid struktuuri ilma andmeteta; andmete hoidmiseks puustruktuuris on mitu mõistlikku võimalust, nagu andmed kõigis tippudes *vs* andmed ainult lehtedes.

Esimesel juhul sobib kahendpuutüüp defineerida koodiga

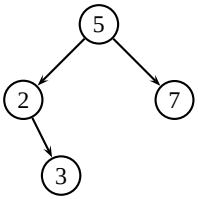
```
data Kahend a
  = Tühi                                .          (255)
  | Tipp a (Kahend a) (Kahend a)
```

Tüübiparameeter definitsioonis (255) näitab puus hoitavate andmete tüüpi. Esimene alternatiiv kirjeldab tühja kahendpuu, teine mittetühjad. Teise alternatiivi konstruktori esimene argument on tüübiparameeter, mis vastab andmele juurtipus. Ülejäänud kaks argumenti on rekursiivsed pöördumised, mis tähendab, et puu need väljad on ise sama tüüpi puud. Neist esimese loeme vasakuks ja teise paremaks haruks.

Näiteks avaldised `Tühi`, `Tipp 0 Tühi Tühi` ja

```
Tipp 5
(
  Tipp 2 Tühi (Tipp 3 Tühi Tühi)
)
(
  Tipp 7 Tühi Tühi
)
```

on kõik korrektsed avaldised tüüpi `(Num a) => Kahend a`. Esimese avaldise väärtus on tühi puu, teise väärtus ühetipuline puu, mille ainsas tipus element 0, kolmanda väärtust kujutab joonis



.

Standardse tüübivaliku puhuks, kui andmed tahetakse paigutada vaid kahendpuu lehtedesse, realiseerib definitsioon

```
data Kahend' a
  = Leht a
  | Harud (Kahend' a) (Kahend' a)
```
. (256)

Tüübiparameetri roll on sama mis definitsioonis (255). Esimene alternatiiv kirjeldab ühetipulised puud, neis on igaühes üks element. Teine alternatiiv vastab hargnevatele puudele nagu definitsioonis (255), kuid nüüd puudub elemenditüüpi väli. Seega mujale kui lehtedesse ei saa andmeid paigutada.

Definitsioon (256) kitsendab muuski mõttes kahendpuude hulka. Nimelt on välistatud tühi puu, mis omakorda tingib selle, et igal tipul, mis pole leht, on 2 alluvat.

Kahendpuu elementide kokkuarvutamine pole sugugi keerulisem kui listilaadse struktuuri korral. Näiteks koodiga (255) defineeritud tüüpide jaoks võiksime elementide loendamise anda koodiga

```
kahendSuurus
  :: Kahend a -> Int'

kahendSuurus (Tipp _ ut vt)
  = 1 + kahendSuurus ut + kahendSuurus vt
kahendSuurus _
  = 0
```
. (257)

Elementide summa leidmiseks mõlema kahendpuutüübi korral võime kirjutada

```
kahendSumma
  :: (Num a)      ,
  => Kahend a -> a
```

```
kahend'Summa
  :: (Num a)        ,
  => Kahend' a -> a
```

```
kahendSumma (Tipp x ut vt)
  = x + kahendSumma ut + kahendSumma vt
kahendSumma _                          ,
  = 0
```
(258)

```
kahend'Summa (Harud ut vt)
  = kahend'Summa ut + kahend'Summa vt
kahend'Summa (Leht x)                  .
  = x
```
(259)

Pangem tähele, et definitsioonid (257), (258), (259) järgivad "jaga ja valitse" strateegiat, kusjuures see on dikteeritud andmestruktuuri ehituse poolt.

Näitena suurema puu konstrueerimisest anname muutuja `täielik` väärtuseks funktsiooni, mis täisarvulisel argumendil annab välja täieliku kahendpuu kõrgusega n. Kahendpuud nimetatakse **täielikuks**, kui tema kõik lehed on samal hierarhiatasemel ja kõigil muudel tippudel on kaks vahetut alluvat. Ilmselt on mittetühja täieliku kahendpuu mõlemad harud 1 võrra väiksema kõrgusega täielikud kahendpuud.

Olgu loodava puu tippudes arvud, mis näitavad vastavast tipust lähtuva alampuu kõrgust. Näiteks avaldise `täielik 2` väärtustamine peaks vastuseks andma `Tipp 2 (Tipp 1 Tühi Tühi) (Tipp 1 Tühi Tühi)`.

Kuna positiivse argumendi korral on otsitava puu alampuud ühesugused, saab abimuutujat kasutades läbi ühe rekursiivse pöördumisega. Koodiks tuleb

```
täielik
  :: Int -> Kahend Int'

täielik n
  = case compare n 0 of
      GT
        -> let
             t = täielik (n - 1)
           in                                .
           Tipp n t t
      EQ
        -> Tühi
      _
        -> error "täielik: neg. argument"
```
(260)

Huvitava asjaoluna märgime, et arvutuse maht koodi (260) puhul kasvab lineaarselt argumendi suhtes (rekursiivsete pöördumiste arv on lineaarne ja igal sammul tehakse ühepalju tööd). Tulemus aga väljendab argumendi suhtes eksponentsiaalse suurusega puud. Ka interaktiivses keskkonnas saadav väljund eksponentsiaalse suurusega, aga see töömaht tuleb sisse alles sõnekujule viimisel, mitte puu ehitamisel.

Saab defineerida ka lõpmatuid puid. Näiteks kood

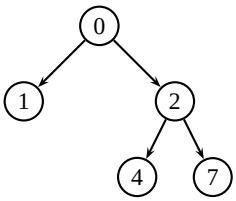
```
lõpmatu
  :: a -> Kahend a'

lõpmatu x
  = let
      t = Tipp x t t .
    in
    t
```

kirjeldab funktsiooni, mis suvalisel argumendil koostab lõpmatu täieliku kahendpuu (igal tipul kaks vahetut alluvat), mille igas tipus on etteantud element.

Ülesandeid

464. Kirjutada avaldis tüüpi `Kahend Int`, mis väljendab joonisel



kujutatud kahendpuud.

465. Kirjutada mõned avaldised, mis väljendavad erineva suurusega kahendpuid tüüpidest, mis defineeritud koodiga (256).

466. Defineerida muutuja `kahend'Suurus` väärtuseks funktsioon, mis võtab argumendiks kahendpuu koodiga (256) defineeritud tüübist ja annab tulemuseks tema elementide arvu.

467. Defineerida muutuja `kõrgus` väärtuseks funktsioon, mis võtab argumendiks kahendpuu deklaratsiooniga (255) defineeritud tüübist ja annab tulemuseks tema kõrguse (tiputasemete arvu).

468. Defineerida puutüüp, mille igal puul on juurtipp, iga puu igal tipul on kas 0 või 2 alluvat ning igas tipus on element.

469. Kirjutada operaatorid ülesandes 468 defineeritud kahendpuu elementide arvu ja summa leidmiseks.

470. Defineerida muutuja `täielik'` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see on mittenegatiivne, siis annab tulemuseks koodiga (255) defineeritud tüüpi täieliku kahendpuu kõrgusega n, kuid andmed tippudes olgu arvud, mis näitavad tipu hierarhiataseme numbrit juurest lugedes.

471. Defineerida muutuja `nummerdus` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see on mittenegatiivne, siis annab tulemuseks kahendpuu koodiga (256) defineeritud tüübist, mille kõik lehed asuvad hierarhiatasemel n ja kus andmed näitavad lehe järjekorranumbrit vasakult lugedes.

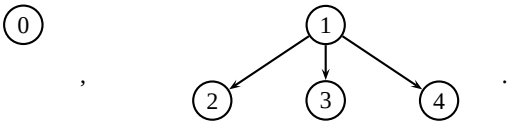
472. Defineerida muutuja `pügaKahend` väärtuseks funktsioon, mis võtab järjest argumendiks täisarvu n ja kahendpuu t koodiga (255) defineeritud tüübist ja annab tulemuseks kahendpuu, mille saab puust t nende osade mahalõikamisel, mis jäävad juurest kaugemale kui n hierarhiataset.

2.2. Mitmerajalised puud. Vastandina kahendpuudele nimetatakse suvaliselt hargnevaid puid **mitmerajalisteks**. Ka mitmerajalise puu juurest erinevad tipud jagunevad harudesse, kuid harusid võib olla suvaline arv. Mitmerajalise puu harud paigutatakse funktsionaalses programmeerimises tavaliselt listi.

Standardne definitsioon on

```
data Mitmeraja a
  = Juur a [Mitmeraja a]
```
. (261)

See, et definitsioonis (261) puudub rekursiivse pöördumiseta alternatiiv, ei tähenda, et puud alati lõpmatuseni hargnevad. Kui harude list on tühi, siis alampuid järelikult pole ja tegemist on lehega, alluvateta tipuga. Nii on korrektsed näiteks avaldised `Juur 0 []` ja `Juur 1 [Juur 2 [], Juur 3 [], Juur 4 []]`, mis väljendavad vastavalt puid



Hargnemine ja mittehargnemine on kirjeldatud ühtsel viisil.

Mitmerajalise puu elementide summa arvutamine on ka peaaegu niisama lihtne kui kahendpuul; sobib kood

```
mitmerajaSumma
  :: (Num a)          ,
  => Mitmeraja a -> a

mitmerajaSumma (Juur x rs)
  = x + sum (map mitmerajaSumma rs).
```

Mitmerajaliste puudega tegelemisel on mugav metsad eraldi tüübina defineerida. Sellisel juhul võib harude kogumit käsitleda metsana, mis viib puu ja metsa loomuliku vastastikrekursiivse definitsioonini. Üks neist võib olla sisse toodud tüübisünonüümina.

Võiksime sisse tuua tüübisünonüümi `Mets` koodiga

```
type Mets a
  = [Mitmeraja a].
```
(262)

Seda nime kasutades võib lihtsustada definitsiooni (261), kirjutades

```
data Mitmeraja a
  = Juur a (Mets a).
```
(263)

Nüüd on uue tüübi definitsioon (263) tüübisünonüümi definitsiooniga (262) vastastikku rekursiivne.

Ülesandeid

473. Defineerida muutuja `mitmerajaSuurus` väärtuseks funktsioon, mis võtab argumendiks mitmerajalise puu ja annab tulemuseks tema elementide arvu.

474. Defineerida puu- ja metsatüübid nii, et metsatüüp oleks uus ja puutüüp sünonüüm. Üldine struktuur peab säilima, liigendkohtade konkreetne ehitus võib olla teistsugune kui ülaltoodud definitsioonide puhul.

475. Kirjutada operaatorid ülesandes 474 defineeritud puude andmeväljade arvu ja summa leidmiseks.

476. Lahendada ülesandega 472 analoogiline ülesanne mitmerajaliste puude jaoks.

477. Kirjutada operaatorid, millega saaks teisendada mitmerajaliste puude ja metsade kahe esituse vahel: üks, mis antud deklaratsioonidega (262) ja (263), ja teine, mis kirjutatud ülesandes 474.

478. Defineerida muutuja `teed` väärtuseks funktsioon, mis võtab argumendiks mitmerajalise puu ja annab tulemuseks sama tüüpi elementidega listide listi, kus listid vastavad teedele juurest lehte läbi alluvussuhete, listi elementideks on teel läbitud tippudes paiknevad andmed.

479. Defineerida muutuja `puuna` väärtuseks funktsioon, mis võtab argumendiks võrdusega tüüpi elementide listide listi ja annab tulemuseks sama tüüpi elementidega mitmerajalise puu, millele ülesandes 478 kirjeldatud operatsiooni rakendamisel saaksime algse listi tagasi. Leitava puu ühegi tipu vahetutes alluvates olevate elementide seas ei tohi olla kordumisi.

2.3. Puustruktuuri esitamine funktsioonide abil. Üldiselt võib puus olla erinevatel alampuudel erinev arv harusid. Kahendpuude korral aga on harude arv alati 2. Kui tegu pole kahendpuudega, kuid harude arv on fikseeritud, siis on võimalik puutüüp esitada funktsioonitüübi, mitte listitüübi kaudu. Funktsiooniga esitatakse harude mets: puu harud on funktsiooni väärtused kõikvõimalikel argumentidel.

Realiseerime selle idee näitena selliste struktuuritüüpide jaoks, mis sisaldavad ka tühja struktuuri. Saame koodi

```
data FPuu a b
  = FTühi
  | FTipp a (FMets a b)

type FMets a b
  = b -> FPuu a b
```
, (264)

mis lühidalt kokkuvõttes ütleb, et F-puu on kas tühi või koosneb ühest komponendist ja F-metsast, kusjuures F-mets realiseeritakse funktsioonina, mis igale andmele teatud teisest tüübist seab vastavusse ühe F-puu.

Andmestruktuuri kujulaad sõltub nüüd argumenditüübist. Näiteks listilaadsed struktuurid tekivad, kui argumenditüüp sisaldab vaid ühe andme. Kui aga argumenditüüp sisaldab parajasti kaks annet, saame kahendpuud. Niisiis saame listid ja kahendpuud ühekorraga ära kirjeldada.

Hea stiili mõttes võtame konkreetsete argumenditüüpidena kasutusele uued tüübid, kus konstruktorite nimed on valitud vastavalt nende rollile.

Listi vahetut alamstruktuuri nimetatakse sabaks, seetõttu olgu ühe andmega tüüp defineeritud koodiga

```
data ListAlam
  = Saba
```
. (265)

Kahendpuu harude tähistamiseks kirjutame analoogiliselt definitsiooni

```
data KahendAlam
  = Vasak      .                                    (266)
  | Parem
```

Süsteemile (264) tuginedes võime nüüd F-listi ja F-kahendpuu määratleda erijuhtudena üldisest F-puust vastavalt deklaratsioonidega

```
type FList a
  = FPuu a ListAlam'                                (267)
```

```
type FKahendpuu a
  = FPuu a KahendAlam.                              (268)
```

F-listid on loomulikus üksüheses vastavuses tavaliste listidega ning F-kahendpuud deklaratsiooniga (255) sisse toodud kahendpuudega. Näiteks listide jaoks saame vastavuse kirjeldada koodiga

```
listFKujule
  :: [a] -> FList a'

fKujultList
  :: FList a -> [a]'

listFKujule (x : xs)
  = FTipp x (\ ~Saba -> listFKujule xs)
listFKujule _                                       ,  (269)
  = FTuhi

fKujultList (FTipp x fxs)
  = x : fKujultList (fxs Saba)
fKujultList _                                       .  (270)
  = []
```

Kuna F-tüübid pole klassis Show, ei saa F-struktuure interaktiivses keskkonnas näidata. Definitsioonide (269) ja (270) korrektsust võib kontrollida operaatorite `listFKujule` ja `fKujultList` järjestrakendamise teel — see peab algse argumentlisti tagasi andma.

Kui koodiga (226) on sisse toodud lõplike tüüpide klass ja lõplike argumenditüüpidega funktsioonide tüübid viidud koodiga (231) klassi Show, saab F-tüüpide kuuluvuse klassi Show automaattuletuse teel tekitada. Et sellega hõlmataks ka F-listid ja F-kahendpuud, peavad vastavad alamstruktuuride nimetusi märkivad tüübid, mis antud koodiga (265) ja (266), olema viidud lõplike tüüpide klassi.

Ülesandeid

480. Defineerida operaatorite `kahendFKujule` ja `fKujultKahend` väärtuseks funktsioonid, mis realiseerivad üksühese vastavuse deklaratsiooniga (255) sisse toodud kahendpuude ja deklaratsiooniga (268) sisse toodud F-kahendpuude vahel.

481. Defineerida muutuja `fListSuurus` väärtuseks funktsioon, mis võtab argumendiks deklaratsiooniga (267) sisse toodud tüüpi F-listi ja annab välja tema elementide arvu. Mitte kasutada vahestruktuure.

482. Defineerida ülesandes 481 kirjeldatud operaatoriga analoogiline operaator `fKahendSuurus` elementide arvu leidmiseks deklaratsiooniga (268) sisse toodud tüüpi kahendpuudes.

483. Tuua sisse kahe parameetriga tüübipere nimega `FPuu'`, mille erijuhuna on võimalik kätte saada sellised kahendpuud nagu defineeritud deklaratsiooniga (256). Defineerida vastav tüübisünonüüm `FKahend'`.

484. Kasutades koodiga (226) sisse toodud lõplike tüüpide klassi, panna F-listide ja F-kahendpuude tüübid kuuluma klassi Show, nii et interaktiivses keskkonnas näidataks nende andmeid sisukalt.

3. Elementideta rekursiivsed andmestruktuurid. Andmestruktuuril ei pea igas tipus olema elementi.

Näiteks koodiga (256) sisse toodud kahendpuudel on elemendid ainult lehtedes.

Mõttekas võib olla isegi selline rekursiivselt defineeritud tüüp, mille üheski andmes pole mingeid elemente.

3.1. Naturaalarvud. Naturaalarvude hulka kuulub arv 0 ning iga muu naturaalarv on saadav mingist naturaalarvust järgmise leidmise (ehk 1 liitmise) teel. Sellest tulenevalt kahe alternatiiviga tüübidefinitsioon, kus üks alternatiividest on paljas konstruktor, teise ainus argument on aga rekursiivne pöördumine, toob sisse tüübi, mille andmed on üksüheses vastavuses kõigi naturaalarvudega.

Olgu definitsioon näiteks

```
data Nat
  = Null      .                                    (271)
  | Järgm Nat
```

See tüüp sisaldab andmed kujul `Null`, `Järgm Null`, `Järgm (Järgm Null)` jne, mis vastavad loomuldasa naturaalarvudele 0, 1, 2 jne.

Seda tüüpi võib kasutada induktiivselt määratletud funktsioonide kirjeldamiseks, kui negatiivsete arvude korral pole lahendus vajalik.

Praktilisest seisukohast pole see ehk siiski kõige parem valik, kuna sisseehitatud täisarvude esitus on ökonoomsem kui naturaalarvude oma (positsioonilise esituse pikkus on võrdeline arvu logaritmiga, kuid naturaalarvutüübis on arvu esituse pikkus võrdeline arvu endaga).

Näiteks liitmise naturaalarvudel saaks sisse tuua koodiga

```
liida
  :: Nat -> Nat -> Nat'

liida (Järgm x) y
  = Järgm (liida x y)
liida _         y    .
  = y
```

Ülesandeid

485. Defineerida koodiga (271) defineeritud tüübi arvude korrutamine.

3.2. Näljased funktsioonid. Kui funktsioonide kaudu üles kirjutatud puustruktuurides loobuda elementidest, on tegu otsekui funktsioonidega, mille väärtusetüüp langeb funktsiooni enda tüübiga kokku. Selline funktsioon võtab ükskõik kui palju argumente ilma tüübiviga andmata. Seda laadi objekte nimetatakse näljasteks funktsioonideks.

Töötame näiteks ümber F-puude definitsiooni (264), kaotades elemenditüübi. Kaotades ühtlasi vastastikrekursiooni ja vahetades konstruktorite ning tüübiparameetri nimed, saame definitsiooni

```
data Näljane a
  = Näljane (a -> Näljane a)·
```
(272)

Arvestades, et andmekonstruktor `Näljane` on vaid tüübiteisendaja, võib nentida, et uue tüübi andmed kujutavad endast funktsioone mingist tüübist sellesse uude tüüpi endasse. Näljase funktsiooni rakendamise operaatori `amps` kirjeldame koodiga

```
amps
  :: Näljane a -> a -> Näljane a·
```
(273)

```
Näljane f ‘amps‘ x
  = f x                                    .
```
(274)

Et väärtusetüüp võrdub funktsioonitüübi endaga, siis saab neile funktsioonidele anda järjest kuitahes palju argumente ühest ja samast tüübist, ilma et tekiks tüübiviga.

Lihtsaim näljase funktsiooni definitsioon on

```
näljane
  = Näljane (const näljane)                ;
```

see paneb muutuja `näljane` väärtuseks näljase funktsiooni, mis annab igal oma argumendil välja iseenda. See tähendab, et see elukas lihtsalt sööb järjest kuitahes palju argumente ja ei tee nendega midagi.

Kui viia näljaste funktsioonide tüübid ka klassi Show koodiga

```
instance Show (Näljane a) where

  show _                                   ,
    = "Anna ampsu! Tahan veel!"
```
(275)

siis saab interaktiivses keskkonnas mängides näha, kuidas `näljane` koos suvalise rea argumentidega (näiteks `näljane` ise; siis `näljane ‘amps‘ 1`; siis `näljane ‘amps‘ 2 ‘amps‘ 5` jne) annab muutumatult vastuseks “Anna ampsu! Tahan veel!”.

Baasjuhtude olemasolu või puudumine tüübidefinitsioonis oluliselt midagi ei muuda. Baasjuhtude korral tekivad tüüpidesse ka andmed, mis pole funktsioonina rakendatavad.

Näiteks võime võtta definitsiooni

```
data Muutuv a
  = Kon
  | Fun (a -> Muutuv a)
```
(276)

koos rakendamisoperaatoriga, mis kirjeldatud koodiga

```
rak
  :: Muutuv a -> a -> Muutuv a’

Fun f ‘rak‘ x
  = f x
_     ‘rak‘ _                            .
  = error "rak: pole funktsioon"
```

Nüüd on konstruktoriga `Fun` ehitatud andmed funktsioonina rakendatavad, kuid konstruktoriga `Kon` ehitatu mitte, tema rakendamine annab täitmisaegse vea. Pangem tähele, et tüübiviga ei anna rakendamisel ka see.

Kui baasjuhte on 1 ja argumenditüübiks võtta selline, mis sisaldab vaid ühe andme, siis hargnemist pole ja näljaste funktsioonide tüüp jäljendab naturaalarvude hulga struktuuri.

Märgime, et operaator `const`, millele oli samuti võimalik kuitahes palju argumente sokutada, ei ole kuidagi näljaste funktsioonidega suguluses. Mainitud operaatori puhul on muutuv argumentide arv võimalik polümorfsuse tõttu; kasutades erineva argumentide arvuga, on tegu operaatori eri väärtustega eri tüüpides. Näljase funktsiooni argumentide arv aga tüüpi ei mõjuta.

Ülesandeid

486. Kirjeldada deklaratsiooniga (272) defineeritud tüübi näljane funktsioon, mille väärtus sõltub argumendist. Kirjutada ümber deklaratsioon (275), nii et erinevus tuleks ka välja.

487. Defineerida naturaalarvutüüp koodiga (276) sisse toodud tüübipere liikmena. Kirjeldada funktsioonid, mis kujutavad uue naturaalarvutüübi loomulikul viisil vanaks ja vastupidi.

6.2.3 Nimelised väljad

1. Väljanimede sissetoomine. Haskellis on võimalus anda andmestruktuuri väljadele nimesid. Väljanimed ehk selektorid tuuakse sisse uue tüübi definitsiooni paremas pooles. Igas alternatiivis tuleb nimed anda kas kõigile väljadele või mitte ühelegi. Nimede panekul kirjutatakse väljanimed oma tüübiga annoteeritult konstruktori järele looksulgudesse ja eraldatakse üksteisest komadega. Alternatiiv jääb seega kujule

$$
\begin{array}{l}
\texttt{c} \\
\texttt{\{} \\
\quad s_1 \texttt{ :: } t_1\texttt{,} \\
\quad \texttt{.........,} \\
\quad s_l \texttt{ :: } t_l \\
\texttt{\}}
\end{array}
\quad , \tag{277}
$$

kus $\mathfrak{c}$ on uus andmekonstruktor, $\mathfrak{s}_i$ on väljade nimed ning $\mathfrak{t}_i$ on vastavalt nende tüübid.

Näiteks isikuandmete kirjete jaoks võime defineerida tüübi `Isik` koodiga

```
data Isik
  = Isik
    {
      eesnimi :: String,
      perenimi :: String,
      sünniaeg :: Daatum,
      sugu :: Sugu
    }
  deriving (Show)
```
(278)

Siin tüüp `Daatum` on toodud sisse deklaratsiooniga (242), tüübi `Sugu` aga võime defineerida näiteks koodiga

```
data Sugu
  = Naine
  | Mees
  | Sega
  deriving (Eq, Show)
```

et keegi end kõrvalejäetuna ei tunneks.

2. Nimeliste väljadega struktuurid. Andmeid nimeliste väljadega tüübist saab kirjeldada sarnase süntaksiga. Signatuuride asemel on võrdused, mille vasak pool on selektor ja parem pool on avaldis vastava välja arvutamiseks.

Näiteks Eesti esimese presidendi isikuandmete kirje saab anda koodiga

```
president
  :: Isik'

president
  = Isik
    {
      eesnimi = "Konstantin",
      perenimi = "Päts",
      sünniaeg = Daatum 1874 Veebruar 23,
      sugu = Mees
    }
```
(279)

Seejuures pole nimeliste väljade omavaheline järjekord oluline. Definitsiooni (279) võib samaväärselt ümber kirjutada näiteks kujul

```
president
  = Isik
    {
      perenimi = "Päts",
      eesnimi = "Konstantin",
      sugu = Mees,
      sünniaeg = Daatum 1874 Veebruar 23
    }
```

kus perekonnanimi on eesnime ees ja sünniaeg on nihkunud lõppu. Väljade nimed välistavad segimineku.

2.1. Nimeliste väljadega struktuuride olemus. Nimelised väljad Haskellis on vaid nn süntaktiline suhkur, mis ei too sisse midagi olemuslikult uut. Kui tüübi $\mathfrak{u}$ definitsioonis on alternatiiv kujul (277), siis konstruktori $\mathfrak{c}$ tüüp on $\mathfrak{t}_1$ `->` ... `->` $\mathfrak{t}_l$ `->` $\mathfrak{u}$, täpselt nagu siis, kui alternatiiv olnuks

$\mathfrak{c}\ \mathfrak{t}_1\ \ldots\ \mathfrak{t}_l$.

Seega töötab ka tavaline viis andmestruktuuri moodustamiseks, kus välju märkivad avaldised antakse järjest konstruktori argumendiks. Sellisel juhul peab väljade järjekord olema sama mis tüübidefinitsioonis.

Näiteks muutuja `president` definitsioon koodis (279) on täiesti samaväärne ka definitsiooniga

```
president
  = Isik
    "Konstantin"
    "Päts"
    (Daatum 1874 Veebruar 23)
    Mees
```

2.2. Selektorid kui muutujad. Nimeliste väljadega definitsiooni puhul on tavapärastel viisidel kasutatavad mitte ainult uued konstruktorid, vaid ka kõik väljanimed. Kui tüübi $\mathfrak{u}$ definitsioonis on alternatiiv kujul (277), siis iga $i = 1, \ldots, l$ korral on selektor $\mathfrak{s}_i$ ühtlasi muutuja tüüpi $\mathfrak{u}$ `->` $\mathfrak{t}_i$ ja tema väärtus on funktsioon, mis annab konstruktoriga $\mathfrak{c}$ moodustatud kirje järgi välja i-nda välja väärtuse.

Nii on definitsioonide (278) ja (279) kontekstis korrektne näiteks rakendamine `perenimi president`, mille väärtus on sõne “Päts”.

Et tüübidefinitsiooniga (278) sisse toodud uusi nimesid ilma nimeliste väljadeta samaväärselt defineerida, peaksime kirjutama definitsioonid

```
data Isik
  = Isik String String Daatum Sugu'

eesnimi (Isik en _ _ _)
  = en                  ,

perenimi (Isik _ pn _ _)
  = pn                  ,

sünniaeg (Isik _ _ sünd _)
  = sünd                ,

sugu (Isik _ _ _ sugu)
  = sugu                .
```

See oleks kohmakas ja programmeerija kaotaks võimaluse muutujaid `eesnimi`, `perenimi`, `sünniaeg`, `sugu` kasutada väljanimedena. Ka automaatselt tuletatav sõneksteisendus töötaks teisiti.

Kui selektor esineb ainult ühes alternatiivis, siis tema väärtuseks olev funktsioon on ainult selle alternatiivi konstruktoriga ehitatud andmestruktuuridel määratud. Teistel argumentidel on tulemuseks $\perp$ (väärtustamine lõpeb veateatega). Kuid sama väljanimi võib esineda tüübidefinitsiooni mitmes alternatiivis, eeldusel et samanimeliste väljade tüüp on sama (viimane nõue on selleks, et selektor oleks korrektselt tüübitav).

Näiteks võiksime kalendripäevatüübi definitsioonis (248) sisse tuua ka selektori `daatum`, mis oleks ühtviisi kasutatav nii gregooriuse kui juuliuse kalendri kuupäevade jaoks. Sobib definitsioon

```
data Kalendripäev
  = Gregooriuse { daatum :: Daatum }
  | Juuliuse { daatum :: Daatum }    .
  deriving (Show)
```

2.3. Polümorfsed selektorid. Kui nimelised väljad tuuakse sisse polümorfse tüübi definitsioonis, saavad ka nemad polümorfseks.

Näitena nimeliste väljadega andmestruktuuride tüübipere kirjeldamisest võib anda koodiga (255) sisse toodud kahendpuutüüpidele uue definitsiooni

```
data Kahend a
  = Tühi
  | Tipp
    {
      juur :: a,
      vh :: Kahend a,
      ph :: Kahend a
    }
```
, (280)

mis toob mittetühjade puude juurtipu elemendi ning vasaku ja parema haru jaoks sisse selektorid `juur`, `vh`, `ph`.

Et nimeliste väljadega andmestruktuuride tüübid on põhimõtteliselt võrdsed selektorideta tüüpidega, kus väljad on konstruktori argumendid tüübidefinitsioonis antud järjestuses, siis kõik varasem kahendpuutüüpe kasutav kood töötab ka definitsiooni (280) korral.

Ülesandeid

488. Anda definitsiooniga (242) sisse toodud kuupäevatüübile mõistlike nimeliste väljadega alternatiivne definitsioon, mis tüüpi kuuluvaid andmeid võrreldes varasemaga ei muuda.

489. Täiendada kahendpuutüüpide definitsiooni (280) nii, et kõigil kahendpuudel oleks lisaks üks lühikese täisarvu tüüpi väli nimega `h`.

3. Väljanimed näidistes. Selektorid võivad kasutust leida ka näidistes. Sellise näidise moodustamine käib samamoodi nagu avaldise moodustamine konstruktori ja komadega eraldatud võrduste loeteluga looksulgudes, kuid võrduste paremates pooltes on vastavaid välju märkivad näidised. Nagu avaldiseski, on väljanimede kasutamisel väljade järjekord vaba.

Näiteks muutujale `kahendSumma` saaks koodis (258) oleva definitsiooni asemel kirjutada definitsiooni

```
kahendSumma (Tipp { juur = x, vh = ut, ph = vt })
  = x + kahendSumma ut + kahendSumma vt
kahendSumma _
  = 0
```
, (281)

kus muutujad `x`, `ut`, `vt` esinevad samas tähenduses kui koodis (258).

On lubatud võrdus ära jätta, kui ta ei seo ühtki vajalikku muutujat.

Näiteks muutuja `kahendSuurus` definitsiooni (257) asemel saaks kirjutada selektoridega samaväärse definitsiooni

```
kahendSuurus (Tipp { vh = ut, ph = vt })
  = 1 + kahendSuurus ut + kahendSuurus vt
kahendSuurus _                                  ,
  = 0
```

kus nimeliste väljadega näidises väli `juur` puudub. Pole vajadust lisada võrdust `juur = _`, kuid see pole ka keelatud.

Haskelli standard ei luba selektore väljaspool looksulgude-komade erisüntaksit väljanimena kasutada; mujal on selektor alati vaid funktsioonitüüpi muutuja. Realisatsioonide laiendused on siin lõdvemad.

Võib tekkida küsimus, miks on definitsioonis (281) muutujaid `x`, `ut`, `vt` on üldse vaja sisse tuua, kui väljanimed võiksid nende osa suurepäraselt ära täita. Hugsi ja GHC laiendused lubavad näiteks definitsiooni (281) asemel kirjutada ka

```
kahendSumma (Tipp { juur, vh, ph })
  = juur + kahendSumma vh + kahendSumma ph
kahendSumma _                                   .
  = 0
```

Siis on võrdusi vaja vaid muutujast erineva kujuga näidise sidumiseks.

4. Värskendamine. Haskell pakub kõigele sellele lisaks veel ühe süntaktilise konstruktsiooni nimeliste väljadega andmestruktuuridega opereerimiseks — värskendamise.

Kui $\mathfrak{e}$ on avaldis, mille väärtus on andmestruktuur nimeliste väljadega tüübist, siis saab sama konstruktoriga moodustatud andmestruktuurid, mis sellest mõne välja poolest erinevad, kirjeldada avaldisega kujul

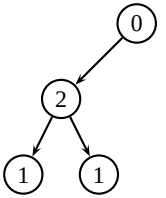
$$\mathfrak{e}\ \texttt{\{}\ \mathfrak{s}_{i_1}\ \texttt{=}\ \mathfrak{a}_1\texttt{,}\ \ \ldots\texttt{,}\ \ \mathfrak{s}_{i_k}\ \texttt{=}\ \mathfrak{a}_k\ \texttt{\}}, \tag{282}$$

kus $\mathfrak{s}_{i_1}, \ldots, \mathfrak{s}_{i_k}$ on avaldisega $\mathfrak{e}$ kirjeldatud andmestruktuuri mingid väljanimed ja $\mathfrak{a}_1, \ldots, \mathfrak{a}_k$ on avaldised, mis kirjeldavad nende väljade väärtusi uues andmestruktuuris. Ülejäänud väljade väärtused on uues andmestruktuuris samad mis originaalses.

Näiteks kui kahendpuutüübid on defineeritud koodiga (280) ja muutuja `täielik` väärtus on etteantud kõrgusega täieliku kahendpuu moodustamise funktsioon (sisse toodud koodiga (260)), siis avaldise

```
(täielik 3) { juur = 0, ph = Tühi }
```
(283)

väärtus on puu



.

Avaldises (282) esinev avaldise `¢` ja värskenduste nimekirja järjestkirjutus on kõrgema prioriteediga isegi funktsioonirakendamisest.

Seetõttu on näites (283) vaja avaldise `täielik 3` ümber sulge.

Ülesandeid

490. Ülesande 489 lahenduse kontekstis kirjutada teek, milles oleksid elementaarsed kahendpuude ehitamise operatsioonid antud muutujate väärtuseks nii, et nende muutujate kaudu oleks võimalik koostada mistahes kahendpuid, kus juures välja `h` väärtus võrduks alati kahendpuu kõrgusega.

6.2.4 Agarad väljad

1. Põhimõte ja kasutamine. Uued konstruktorid on vaikimisi laisa väärtustamisega. See tähendab, et väärtustamise käigus on võimalik olukord, kus andmestruktuur on määratud (st konstruktor, millega ta moodustatakse, on olemas), samal ajal kui väljad on väärtustamata. Kui on vaja andmestruktuuri väärtustamist — näiteks agara funktsiooni argumendis —, siis ei too see kaasa tema komponentide väärtustamist, vaid ainult struktuuri väljaarvutamist kuni moodustava konstruktori ilmumiseni. Struktuur võib tervikuna olla normaalne, samal ajal kui mõni tema komponent on $\perp$.

Kui välju on vaja agaralt väärtustada, siis ühe võimaluse selleks annab muidugi operaator `$!`. Kuid selle operaatori tihe lisamine koodi on tülikas ja muudab koodi raskemini loetavaks.

Andmestruktuuride tüübi defineerimisel võib olla juba selge, et mõne konstruktoriga moodustatud struktuuride sihipärasel kasutamisel on teatavat välja alati vaja väärtustada. Selliseks puhuks on Haskellis võimalus lisada sümbol `!` selle välja tüübi ette *data*-deklaratsioonis; rohkem kui ühest nimest koosnev väljatüüp tuleb seejuures asetada sulgudesse. Hüüumärgiga varustatud väljad väärtustatakse alati kohe andmestruktuuri loomisel, koos moodustava konstruktori ilmumisega. Neid välju nimetatakse agarateks.

Näiteks võib arvata, et koodiga (242) defineeritud kuupäevatüübi väljad on kõik alati olulised, niipea kui kuupäev kui tervik on oluline. Osaliselt väärtustatud kuupäeval ei ole mõtet. Seega võib definitsiooni (242) asendada definitsiooniga

```
data Daatum
  = Daatum !Integer !Kuu !Int,                    (284)
  deriving (Show, Eq, Ord)
```

kus kõik väljad on deklareeritud agaraks.

Märk `!` ei muuda välja tüüpi, vaid ainult konstruktori väärtustamisstrateegiat selle välja suhtes. Hüüumärk esineb ainult uue tüübi definitsioonis, mitte näiteks seda tüüpi sisaldavates tüübiannotatsioonides ega signatuurides. Välja muutmisel agaraks pole mujal koodis vaja mingeid muudatusi teha, kui just kood tema laiskust sisuliselt kasutada ei püüa.

Näiteks senine kuupäevatüüpi kasutav kood läheb ühtviisi läbi nii definitsiooni (242) kui ka definitsiooni (284) puhul.

Haskelli teegis on agarate väljadega realiseeritud näiteks ratsionaal- ja kompleksarvutüübid. Agaraks võib märkida meelevaldse hulga välju.

Andmete hulgana vaadeldes on agarate väljadega tüübid pisut teistsuguse ehitusega kui üdini laiskade konstruktoritega tüübid: neis puuduvad struktuurid, kus agar väli võrdub bottomiga (sellist struktuuri ei moodustu).

Tüübidefinitsiooni (284) korral on avaldised `Daatum 2000 undefined 1` ja `undefined` sama väärtusega $\bot$, kuid originaaldefinitsiooni (242) korral nende väärtused erinevad.

Ülesandeid

491. Anda koodiga (248) defineeritud tüübile uus definitsioon, mille korral konstruktorid oleks agara väärtustamisega.

2. Lõplike andmestruktuuride tüübid. Kui agaraks deklareerimist mitte kasutada, siis kuuluvad rekursiivselt defineeritud tüüpi automaatselt ka lõpmatud ja osalised andmestruktuurid. Deklareerides struktuuritüübi definitsioonis rekursiivset pöördumist sisaldava tüübiga välja agaraks, on sealt hargnev lõpmatu struktuur välistatud.

Näiteks saame nii defineerida tüübi, mis koosneb listilaadsetest struktuuridest, mis on kõik lõplikud. Sobib deklaratsioon

```
data LõplikList a
  = Kõik                    .                    (285)
  | a ‘Ja‘ !(LõplikList a)
```

Tühja listi märgib konstruktor `Kõik`, ühe elemendi lisamist konstruktor `Ja`. See näide illustreerib ühtlasi konstruktori infiksset rakendust oma definitsioonis.

Et infikskuju `‘Ja‘` oleks konstruktoriga `:` sarnase prioriteedi ja assotsiatiivsusega, lisame ka infiksdeklaratsiooni

```
infixr 5 ‘Ja‘.
```

Nüüd saame seda tüüpi “liste” kirja panna avaldistega nagu näiteks

```
1 ‘Ja‘ 2 ‘Ja‘ 3 ‘Ja‘ Kõik.                          (286)
```

Lisades deklaratsioonile (285) rea

```
deriving (Show),
```

võib juhtuda, et paljastame Hugsi ja GHC loojate erineva tõlgenduse klassi Show kuuluvuse automaatse tuletuse detailidest Haskelli keeledefinitsioonis. Et tulemus oleks ühtmoodi loetav oodataval viisil, st sidesõna “ja” esineks listi elementide vahel, võiks loobuda automaatsest tuletusest ja kirjutada oma esindajadeklaratsiooni

```
instance (Show a) => Show (LõplikList a) where

  showsPrec n l as
    = let
        loend (x ‘Ja‘ xs)
          = showsPrec n x . (" ja " ++) . loend xs
        loend _
          = ("kõik)" ++)
      in
      ’(’ : loend l as
```

Nüüd annab avaldise (286) väärtustamine interaktiivses keskkonnas vastuse “(1 ja 2 ja 3 ja kõik)”.

Lõpmatut ega osalist listi koodiga (285) defineeritud tüübist kirja panna ei saa. Avaldis, mis ilma agarate väljadeta annaks väärtustamisel lõpmatu struktuuri, annab nüüd tühja lõpmatu tsükli — struktuuri ei teki, väärtus on $\bot$.

Näiteks kirjutame koodiga (116) defineeritud operaatori `lõplik` definitsiooni ümber selliselt, et argumendiks oleks küll tavaline list, kuid tulemus oleks deklaratsiooniga (285) sisse toodud lõpliku listi tüüpi. Signatuur oleks

```
lõplik
  :: [a] -> LõplikList a'
```

definitsioon ise

```
lõplik (x : xs)
  = x ‘Ja‘ lõplik xs
lõplik _
  = Kõik
```

Nüüd kui mingi avaldise `l` väärtus on lõplik list, siis avaldise `lõplik l` väärtustamise tulemuseks on sama listi väljendav struktuur uuest listitüübist, lõpmatu listi puhul aga jääb arvutus lõpmatusse tsüklisse midagi välja andmata.

Lõpliku listi tüüpi struktuure saab väärtustada ainult kas lõpuni või üldse mitte. Näiteks avaldise `const 0 $! (lõplik [1 .. 10000])` väärtustamine ehitab 10000-elemendilise struktuuri; võrdluseks, `const 0 $! [1 .. 10000]` väärtustamisel väärtustatakse list ainult esimese koolonini.

Ülesandeid

492. Defineerida kahendpuutüüp, millesse kuuluvad parajasti kõik vasakult lõplikud (st kõik vasakute alluvate jadad on lõplikud) kahendpuud.

493. Defineerida lõplike mitmerajaliste puude tüüp.

6.2.5 Tüüpide ümbernimetamine

1. Ümbernimetamise süntaks. Nimetame mähistüüpideks selliseid tüüpe, mille definitsioonis on täpselt üks alternatiiv ja selle konstruktoril täpselt üks argument. Selline tüüp sisaldab täpselt ühe väljaga andmestruktuure, mis kõik luuakse sama konstruktoriga — järelikult tekib üksühene vastavus andmestruktuuride tüübi ja väljatüübi andmete vahel (kui struktuuritüübi bottom kõrvale jätta).

Näiteks deklaratsioonid (244) ja (245) defineerivad mähistüübi.

Haskellis on olemas võimalus mähistüüpide definitsioonis kirjutada võtmesõna **`data`** asemel võtmesõna **`newtype`**, mis kannab tavalisest tüübidefinitsioonist pisut erinevat tähendust. Sellist tüübidefinitsiooni kutsutakse tüübi ümbernimetamiseks.

Niisiis on ümbernimetamine üldiselt kujul

```
newtype u
  = c v    ,
```
(287)

kus u koosneb uue tüübi nimest koos argumente tähistavate muutujatega (peab olema täisargumenteeritud), c on uus andmekonstruktor ja v tüübiavaldis, mis võib sisaldada u-s sisse toodud muutujaid. Ainsale väljale võib anda ka nime. Sarnaselt tavaliste tüübidefinitsioonidega võib deklaratsioonile (287) lisada *deriving*-klausli klassikuuluvuste automaattuletuseks.

Näiteks deklaratsioonid (244), (245) on ümbernimetamisega defineerides vastavalt

```
newtype Aasta
  = Aasta Integer'

newtype Päev
  = Päev Int.
```
(288)

Ümbernimetamine töötab ka muutujast keerulisemate väljatüüpide korral.

1.1. Rekursiivne ümbernimetamine. Ümbernimetamisdeklaratsioon võib olla rekursiivne, st ainsa välja tüüp kirjeldatud uue tüübi enda kaudu.

Sellisel juhul ei ole ümbernimetamisel tekkiv uus tüüp mingi juba olemasoleva tüübi kloon, vaid sisuliselt uus tüüp, kuna ka väljatüüpi märkiv tüübiavaldis ise ei oma ilma selle deklaratsioonita tähendust.

Näiteks varasemalt antud näljaste funktsioonide definitsiooni (272) võiks asendada ümbernimetamisega

```
newtype Näljane a
  = Näljane (a -> Näljane a).
```
(289)

Kõik varasema deklaratsiooniga (272) sisse toodud tüüpide jaoks kirjutatud kood võib üleminekul deklaratsioonile (289) jääda endiseks.

Ülesandeid

494. Kirjutada ümbernimetamine, mis rekursiivselt defineerib tüübi, millesse kuuluvad andmed on üksüheses vastavuses teatud elementidtüübiga mittelõplike listidega.

2. Ümbernimetamise semantiline eripära. Tegu on mõneti vaheastmega *data*- ja *type*-deklaratsiooni vahel. Deklaratsiooniga (287) antakse vanale tüübile `ʊ` uus vorm `ʉ`, mis oleks justkui täiesti uus tüüp, kuid tegelikult on uus vorm nähtav vaid programmeerijale koodis ja masinale tüübianalüüsi etapil. Andmetega opereerimise ajaks (st programmi jooksutamisel) heidetakse ümbernimetatud tüüpide andmekonstruktorid minema. Tüübi `ʉ` andmete asemel jooksevad vastavad tüübi `ʊ` andmed, otsekui oleks deklaratsiooni (287) asemel olnud sünonüümideklaratsioon

```
type ʉ
  = ʊ .
```

Tulenevalt ümbernimetatud tüüpide andmekonstruktorite eemaldamisest väärtustamise etapil on neil konstruktoritel mõned üllatavad omadused, mida võib olla kasulik programmeerimisel arvestada.

2.1. Agar väärtustamine. Andmekonstruktori eemaldamine väärtustamise ajaks paistab teoreetilises plaanis välja kui konstruktori agarus. Kuna täitmisajal seda konstruktorit näha pole, ei ole ümbernimetatud tüüpi avaldise väärtustamisel võimalik väärtuse normaalsuse üle otsustada moodustava konstruktori väljailmumise järgi, mistõttu väärtustamine jätkub sujuvalt konstruktori alla jääva alamavaldisega ja otsus toimub lähtuvalt selle väärtuse normaalsusest.

Selle tõttu pole Haskellis lubatud tüübi ümbernimetamisel välja agaraks märkida. Väli on agar niikuinii, sel märkel poleks mõtet.

2.2. Näidise sobitumine bottomiga. Andmekonstruktori eemaldamise teine efekt ilmneb näidisesobitusel: ümbernimetatud tüübi andmekonstruktoriga moodustatud näidis võib sobituda ka bottomiga. Kuna seda konstruktorit pole näha, ei tehta tema abil moodustatud näidise sobitamisel otsust selle konstruktori enda juures, vaid see ülesanne kandub üle näidise siseosale.

See pole sama mis laisa näidise käitumine: laisk näidis ei delegeeri otsustust sissepoole, vaid musta kastina sobitub iga avaldisega.

Sobitumist bottomiga võib näha kui avaldiste väärtustamise lõppkujude tavatut käsitlust, millest lähtub näidise struktuuri klappimine või mitteklappi-

mine. Kui tavaliste tüüpide puhul on bottomi lõppkuju ilma konstruktorita ja seda ka agarate väljade korral, siis ümbernimetamisega loodud tüüpidel loeme bottomi lõppkujuks konstruktoriga kuju.

Deklaratsiooni (288) puhul annavad definitsioonid

```
(x , Aasta y)
  = (1 , undefined)
```
(290)

ja

```
(x , Aasta y)
  = (1 , Aasta undefined)
```
(291)

mõlemad muutuja `x` väärtuseks 1, sest `undefined` ja `Aasta undefined` on samaväärsed, mõlemal juhul sobitatakse näidis `y` avaldise `undefined` vastu ja see õnnestub. Samas definitsioonid

```
(x , Aasta [])
  = (1 , undefined)
```

ja

```
(x , Aasta [])
  = (1 , Aasta undefined)
```

annavad `x` väärtuseks $\bot$, sest mõlemal juhul sobitatakse näidis `[]` avaldise `undefined` vastu ja see ebaõnnestub.

Võrdluseks, originaaldefinitsiooni (244) korral toimub kõik nagu tavaks lõviosa konstruktorite puhul: deklaratsioon (290) annab `x` väärtuseks $\bot$, (291) aga 1.

Kui aga aastatüübi definitsioon kirjutada agara väljaga kujul

```
data Aasta a
  = Aasta !a'
```

siis definitsioonid (290) ja (291) annavad mõlemad `x` väärtuseks hoopiski $\bot$. Kuigi ka siin on `undefined` ja `Aasta undefined` samaväärsed, on näidises konstruktor alles, mistõttu viga tuleb välja ja sobitus ebaõnnestub.

3. Ümbernimetamise otstarve. Tüüpide ümbernimetamised on vajalikud selleks, et võimaldada efektiivsemat mähistüüpidesse kuuluvate andmetega opereerimist. Võit efektiivsuses saavutatakse konstruktorist loobumisega arvutuste käigus. Seetõttu on üldjuhul soovitav mähistüübid just ümbernimetamise teel sisse tuua.

6.2.6 Veidrad tüübid

Oleme nüüd näinud mitmesuguseid tüüpe, mis kõik võrdlemisi mõistuspärased. Ometi pole see ligilähedaseltki Haskelli võimaluste piir. Lõpetuseks teeme tutvust tüüpidega, mis võivad lugeja senist intuitsiooni funktsionaalse programmeerimise võimaluste kohta eri viisidel korrigeerida.

1. Rekursiivselt defineeritud tüübifunktsioonid. Rekursiivne tüübidefinitsioon, mille vasakus pooles esineb konstruktori järel tüübimuutujaid, defineerib uue tüübikonstruktori väärtuseks tüübifunktsiooni. Kui aga see tüübikonstruktor on paremas pooles kõikjal rakendatud samamoodi kui vasakus pooles, siis sisuliselt defineeritakse siiski iga tüüp eraldi.

Näiteks deklaratsioonis (261) on defineeritav tüübikonstruktor `Mitmeraja` nii vasakul kui paremal rakendatud ühtmoodi argumendile `a`. Ka sama tüübiperet alternatiivselt defineerivate deklaratsioonide (262) ja (263) vastastikrekursioon avaldab defineeritava tüübikonstruktori sama tüübikonstruktori kaudu samal argumendil.

Samuti näiteks kood (264) avaldab defineeritava tüübikonstruktori `FPuu` argumentidel `a` ja `b` sama tüübikonstruktori kaudu samadel argumentidel ja samas järjekorras, mis tähendab, et sisuliselt on iga F-puutüüp määratletud omaette.

Teist laadi näiteid meil esinenud ei ole.

Ei ole keelatud rekursiivses kasutuses argumente teisiti panna. Kuid selliste tüüpidega seotud muutujate rekursiivsel defineerimisel on vaja lisada tüübisignatuur, sest automaatne tüübituletus vaikimisi eeldab rekursiivse pöördumise olevat algse väljakutsega sama tüüpi.

Oleme harjunud, et listis peavad kõik elemendid olema sama tüüpi. Kuid deklaratsioon

```
data VList a b
  = VTühi
  | VKons a (VList b a)
  deriving (Show)
```
(292)

toob sisse kahest parameetrist sõltuva tüübipere `VList`, mille tüüpidesse kuuluvad listilaadsed andmestruktuurid, mille elemendid on mingit kaht tüüpi vaheldumisi.

Kirjutame ka infiksdeklaratsiooni

```
infixr 5 `VKons`,
```

et muuta konstruktori infiksne kasutus sarnaseks listikonstruktori omaga. Siis on igati korrektsed näiteks avaldised `1 ‘VKons‘ ’A’ ‘VKons‘ VTühi` ja `True ‘VKons‘ "Hai!" ‘VKons‘ False ‘VKons‘ VTühi`.

V-listi elementide arvu leidva funktsiooni defineerib kood

```
vListSuurus (x ‘VKons‘ xs)
  = 1 + vListSuurus xs
vListSuurus _
  = 0
```
, (293)

millele tasub lisada signatuur

```
vListSuurus
  :: VList a b -> Int
```
. (294)

Ilma signatuurita (294) tuletataks deklaratsiooniga (293) defineeritud muutujale `vListSuurus` tüüp, kus argumenditüüp oleks `VList a a`. See tähendab, et `vListSuurus` oleks rakendatav vaid sellistele V-listidele, mille elemendid on kõik sama tüüpi.

On võimalik saada isegi selliseid listilaadseid struktuure, mille kõik elemendid on erinevat tüüpi. Selleks anname defineeritava tüübikonstruktori argumendiks rekursiivsel pöördumisel midagi keerulisemat kui muutuja. Näiteks deklaratsioon

```
data Tabel a
  = Lõpp
  | Järg a (Tabel [a])
  deriving (Show)
```
(295)

tekitab listilaadsed andmestruktuurid, mille esimene element on suvalist tüüpi, teine element on seda tüüpi elementidega list, kolmas element selliste elementidega listide list jne.

Infiksdeklaratsiooni

```
infixr 5 ‘Järg‘
```

korral on võimalik näiteks avaldis

```
1 ‘Järg‘ [1, 2] ‘Järg‘ [[1, 2], [3]] ‘Järg‘ Lõpp
```
.

Deklaratsioon

```
tabelPikkus (x ‘Järg‘ xs)
  = 1 + tabelPikkus xs
tabelPikkus _
  = 0
```
(296)

ilma signatuurita isegi ei kompileeru; tuleb lisada signatuur

```
tabelPikkus
  :: Tabel a -> Int
```

Deklaratsioon (296) annab muutuja `tabelPikkus` väärtuseks funktsiooni, mis deklaratsiooniga (295) sisse toodud tabelitüüpi argumendi järgi leiab tema elementide arvu. Pangem tähele, et tegu pole elementide arvuga "tabelisse" paigutatud listides, vaid "tabelisse" paigutatud objektide endi arvuga.

Ülesandeid

495. Defineerida muutuja `mapVList` väärtuseks funktsioon, mis võtab järjest argumendiks kaks funktsiooni f ja g ning koodiga (292) defineeritud tüüpi V-listi l, mille elemendid on vaheldumisi f argumenditüüpi ja g argumenditüüpi, ja annab tulemuseks niisama pika V-listi, mille elemendid on saadud vastavatest l elementidest vaheldumisi funktsioonide f ja g rakendamisel.

496. Defineerida muutuja `takeTabel` väärtuseks funktsioon, mis on nagu `take`, kuid töötab listide asemel deklaratsiooniga (295) sisse toodud tabelitel.

497. Vastavalt signatuurile

     ```
     korratabelid
       :: Tabel Integer
     ```

     kus tüübikonstruktor `Tabel` on sisse toodud deklaratsiooniga (295), defineerida muutuja `korratabelid` väärtuseks tabel, kus esimene element on 1, teine on list arvudega 1 kuni 10, kolmas on nende arvude korrutustabel listide listina, neljas on nende arvude kolmekaupa korrutiste tabel jne.

498. Koodiga (296) defineeritud operaatoriga `tabelPikkus` saab arvutada tabeli elementide arvu, kus elemendi piiritlemisel käsitletakse kogu struktuuri kui listi. Loomulikumgi võimalus oleks piiritleda element kui selline komponent, mille tüüp võrdub esimese elemendi (nüüd eelmises tähenduses) tüübiga. Leida võimalus defineerida muutuja `tabelSuurus` väärtuseks funktsioon, mis võtab argumendiks Integer-tüüpi elementidega tabeli ja annab välja tema selliste elementide arvu.

2. Funktsionaalse paradigma loomulik tsükkel. Oleme oma ringkäigul läbi funktsionaalse programmeerimise võimaluste näinud mõndagi hämmastavat. Nüüd on käes viimane punkt, mis võiks kõigele panna krooni.

2.1. Eneselerakendatavad funktsioonid. Meenutagem näljaseid funktsioone, mille tüüp võrdus nende väärtusetüübiga. Niisama hästi võib kirjutada tüübidefinitsiooni, mis paneb funktsiooni tüübi võrduma funktsiooni argumenditüübiga.

Näljased funktsioonid sai sisse toodud deklaratsiooniga (272) ja hiljem paremini deklaratsiooniga (289).

Mehhaanilise liigutusega võib koodis (289) vahetada argumendi- ja väärtusetüübi.

Saame deklaratsiooni

```
newtype Fix a
  = Fix (Fix a -> a)
```
(297)

Kirjutame ka vastava rakendamisoperaatori koodiga

```
Fix f `rak` x
  = f x
```
(298)

signatuur on

```
rak
  :: Fix a -> Fix a -> a
```
(299)

Definitsioon (298) on analoogne näljaste tüüpide jaoks kirjutatud rakendamisoperaatori `amps` definitsiooniga (274). Tulenevalt argumendi- ja väärtusetüübi vahetusest on ka signatuuris (299) võrreldes muutuja `amps` signatuuriga (273) teise argumendi tüüp ja väärtusetüüp kohad vahetanud.

Deklaratsiooniga (297) sisse toodud tüüpi kuuluvad objektid pole enam näljased funktsioonid, vaid midagi hoopis kummalisemat ja arusaamatumat. Need funktsioonid võtavad argumendiks iseendaga sama tüüpi funktsiooni ja annavad selle järgi välja mingi andme mingist suvalisest tüübist. Muuhulgas saab neid kummalisi funktsioone rakendada ka iseendale.

Matemaatikas on funktsiooni rakendamine iseendale lubamatu, sest see viiks loogiliste vasturääkivuste, paradoksideni. Võrdus nagu see paistab definitsioonist (297) — hulk X võrdub funktsioonide hulgaga sellest hulgast kuhugi teise hulka A — on võimatu juba suuruslikel kaalutlustel (tõestus diagonaliseerimisega näitab, et kui hulgas A on vähemalt 2 elementi, siis funktsioonide hulk on alati suurem). Haskellis saab seda lubada pisukeste erinevuste tõttu funktsioonide olemuses võrreldes matemaatikaga.

2.2. Eneselerakendamise operaator. Võime sisse tuua isegi eneselerakendamise operaatori koodiga

```
eneserak f
  = f ‘rak‘ f˙
```
(300)

On tunda vaheleruttavat küsimust: milleks see absurdimaiguline mäng? Kas tasub mingitki praktilist kasu kõigest sellest oodata? Ega vist, kuna juba näljased funktsioonid olid pigem vaid huvitav mõttetreening?…

Kummaliste funktsioonidega tasub siiski veel pisut mängida.

2.3. Eneselerakendamise eneselerakendamine. Pangem tähele eneselerakendamise operaatori signatuuri

```
eneserak
  :: Fix a -> a˙
```

Operaatori `eneserak` tüüp on sama, mida näeme tüübidefinitsioonis (297) andmekonstruktori `Fix` all. Seega koos mähkiva andmekonstruktoriga `Fix`, mis ju sisu ei muuda, oleks eneselerakendamise tüüp `Fix a`.

Nüüd kuna eneselerakendamine kujul `Fix eneserak` on tüüpi `Fix a`, on võimalik ka eneselerakendamist ennast tüübikorrektselt enesele rakendada. Teisi sõnu, avaldis

```
eneserak (Fix eneserak)
```
(301)

on tüübikorrektne. Järelikult on mõtet rääkida selle avaldise väärtusest.

Mis aga on avaldise (301) väärtus?

Siia sobib vahepalaks järgmine ajalooline lugu 1770.–80. aastatest, kui keemia oli teadusena alles lapsekingades. Põlemisnähtuste seletamisel oli veel valdavaks flogistoniteooria, mis seletas põlemist erilise tuleaine — flogistoni — eraldumisega. Flogistoni täpsema olemuse ümber käisid vaidlused, kuid mainitud ajaks oli õpitud saama vesinikku ja paljud keemikud arvasid, et vesinik ongi flogiston.

Tõenäoliselt nõudis julgust esmakordne katse arvatavat puhast tuleainet põlema panna, kuid ka see oli tolleks ajaks tehtud. Teati, et see gaas põleb hästi. (Erinevalt eneselerakendamise eneselerakendamisest, mida võib julgesti interaktiivses keskkonnas katsetada, kulus arvatava tuleaine süütamise puhul ettevaatlikkus kindlasti ära, sest õhuga segi vesinik võib plahvatada.) Kuid siis märgati ka, et selle gaasi põlemisel tekivad mingi vedeliku piisad.

Teadlased nähtavasti põlesid uudishimust teada saada, mis vedelik see on. (Nimetagem siin peamisi tegijaid ka nimepidi — kõigepealt Macquer, siis Cavendish ning mõned aastad hiljem Lavoisier ja Laplace.) Ja kui see selgus, siis… Nad võisid varem ühte või teist oletada, aga kas ka seda?

Interaktiivses keskkonnas avaldise (301) väärtustamine jääb tühjalt mõtlema. Järelikult väärtus peab olema bottom.

On see siis midagi erilist? Oleme tühjalt mõtlemajäämist varemgi kogenud!

See ei olekski midagi erilist, kuid pangem tähele, et tühjalt mõtlemajäämine näitab, et arvutus on jäänud tsüklisse.

Arvasime, et funktsionaalses keeles ilma rekursioonita tsüklilist protsessi programmeerida on võimatu. Kuid ei muutuja `eneserak` definitsioon ega ka tema kasutatava muutuja `rak` definitsioon pole rekursiivne ning kumbki definitsioon ei kasuta ka muid, mujal rekursiivselt defineeritud muutujaid!

Oleme avastanud funktsionaalse arvutusmootori oma, loomuliku tsükli. See kasvab üles sealt, kus matemaatikas tekivad paradoksid.

Demonstreerimaks seda ning ühtlasi põhimõttelist erinevust matemaatikast, kus eneselerakendamine loogiliselt võimalik ei ole, kodeerime Haskellis kummaliste funktsioonide kaudu Russelli (habemeajaja) paradoksi. Võtame seda predikaatide keeles: nimetame impredikaabliks predikaatidel töötavat predikaati, mis iseendal ei kehti; kas impredikaablus ise on impredikaabel?

Impredikaabluse, st enesel mittekehtivuse, defineerime koodiga

```
impr p
  = not (eneserak p)'
```
(302)

signatuur on seejuures

```
impr
  :: Fix Bool -> Bool˙
```

Lastes nüüd arvutada impredikaabluse impredikaablust ehk väärtustada avaldist `impr (Fix impr)`, jääb arvuti tsüklisse. Paradoksi pole, väärtus on bottom.

Ainus erinevus varasemast, loomuliku tsükli põhinäitest koodiga (300) ja (301) on lisandunud `not`.

Bottomi väljatulekut eneselerakendamise eneselerakendamisel võib lihtsasti ka katsetamata mõista ja koguni kahe mõttekäiguga.

Kõigepealt, avaldise `eneserak (Fix eneserak)` tüüp on `a`, suvaline tüüp. Et eneselerakendamine ei too sisse vähimatki infot tüübi kohta, siis eneselerakendamise eneselerakendamine peab töötama täiesti universaalselt, tüübist sõltumata. Ainus anne, mis on kõigis tüüpides ühesugune, on bottom. (Ja et näidisesobitus ei saa ebaõnnestuda ning veateadet pole kirjutatud, siis jääbki üle ainult tsükkel.)

Arutleda saab ka otseselt. Erinevuse `eneserak` ja `Fix eneserak` vahel võib unustada ja rääkida ainult eneselerakendamise eneselerakendamisest. Eneselerakendamise operaatori rakendamine argumendile ƒ kirjutatakse ümber kui ƒ rakendamine ƒ-le. Kui aga ƒ on ka ise eneselerakendamise operaator, siis tekib eneselerakendamise operaatori rakendamine eneselerakendamise operaatorile ehk enesele ja ollaksegi alguses tagasi.

2.4. Püsipunktikombinaatori defineerimine. Niisiis oleme leidnud, et tsükkel kui selline ei ole funktsionaalse arvutuse jaoks üldsegi võõras lisand.

Kuid mis kasu on tühjast tsüklist? Vesiniku põlemist kasutades saab vähemalt keskkonnasäästlikke autosid ehitada…

Tsükli väljatulek võiks viia mõttele, et ehk pole see ainus tsükkel, mis rekursioonita kirjeldatav on. Ehk on võimalik samamoodi kirjeldada ka sisukaid tsüklilisi andmeid?

Osutub, et nii on võimalik kirjeldada üldse kõik andmed mis ka rekursiooniga. Selles veendumiseks piisab kirjeldada uuesti ja ilma rekursioonita püsipunktikombinaator `fix` samaväärselt tema varasema definitsiooniga (209) või (212), sest püsipunktikombinaatori kaudu avaldusid kõik tsüklilised protsessid.

Teame, et `eneserak` väärtus on funktsioon, mille väärtusetüüp võib olla suvaline. Järelikult tema komponeerimisel (järjestrakendamisel) suvalise funktsiooniga, mille argumendi- ja väärtustüüp on võrdsed, saame sama tüüpi funktsiooni. Seega võime `eneserak` asemele, kus iganes ta ka esineb, panna ƒ `. eneserak`, kus ƒ väärtus on mingi funktsioon, mille argumendi- ja väärtusetüüp on võrdsed.

Russelli paradoksi kodeerimisel juba tegime seda komponeerimist konkreetse funktsiooniga — loogilise eitusega —, sest definitsioon (302) on argumendivabas stiilis samaväärselt ümber kirjutatult

```
impr
  = not . eneserak
```

Loogilise eituse argumendi- ja väärtusetüüp on võrdsed tõeväärtusetüübiga.

Teeme osutatud asenduse eneselerakendamise eneselerakendamise avaldises (301). Kuna juurde tuleb üks suvaline funktsioon, millega eneselerakendamist komponeeritakse, siis kirjutame definitsiooni, kus see figureerib argumendina. Saame

```
fix f
  = let
      h = f . eneserak .
    in
    h (Fix h)
```
(303)

See `fix` ongi püsipunktikombinaator.

Lugeja võib selles katseliselt veenduda, kirjutades rekursiivseid definitsioone `fix` abil ümber. Kõik töötab nagu püsipunktikombinaatori definitsiooni (209) puhul.

Absurd on kasulikult tööle pandud.

Sarnase aruteluga nagu bottomi ilmumist eneselerakendamise eneselerakendamisel võib mõista ka deklaratsiooniga (303) defineeritud püsipunktikombinaatori toimimist.

2.5. Viimane tõdemus. Lõpetuseks tuleb tõdeda, et rekursioonist siiski täiesti lahti saada ei õnnestunud, sest tüübitaseme rekursioon jäi alles tüübidefinitsiooni (297).

Täielikult õnnestuks rekursioonist vabaneda tüüpideta funktsionaalsetes keeltes.

Ülesandeid

499. Saada aru koodiga (303) defineeritud püsipunktikombinaatori toimimisest. Miks töötavad püsipunktikombinaatori kaudu kirjeldatud tsüklilised arvutused definitsiooni (303) korral mitte kiiresti nagu definitsiooni (212) korral, vaid aeglaselt nagu (209) korral?

500. Kirjutada püsipunktikombinaatori definitsioon (303) samaväärselt ümber lühemalt, kusjuures mitte kasutada muutujast keerulisemaid avaldisi korduvalt ja mitte kasutada rekursiooni.

Järelsõna

Käesolev raamatuke täitis oma laiema eesmärgi, kui lisaks tudengile vajalike oskuste omandamisele tekkis ka sügavam huvi funktsionaalse programmeerimise vastu. See on vaimustav arvutiteaduse valdkond, mis sisaldab palju rohkem põnevaid võimalusi kui see õpik autori parimagi tahtmise korral suutis näidata. Kogu see raamat on ju, nagu ta pealkirigi ütleb, vaid sissejuhatus!

Täiendav lugemisvara

Õpikud

[1] Hutton, Graham: *Programming in Haskell.* Cambridge University Press, 2007.

Lühike, peamiselt näidete varal õpetav sissejuhatus funktsionaalsesse programmeerimisse.

[2] Thompson, Simon: *Haskell: the Craft of Functional Programming.* 2nd ed., Addison-Wesley, 1999.

Kolmest mahukaim raamat, mitmekülgne, nii algajale kui ka edasijõudnule mõeldud peatükkidega õpik. Käesolev raamat on lähenemise poolest neist kolmest õpikust sarnaseim just sellele (kuid sarnasus pole kuigi suur).

[3] Bird, Richard: *Introduction to Functional Programming Using Haskell.* 2nd ed., Prentice Hall, 1998.

Mahult keskmine, lähenemine suures osas praktilistest probleemidest juhitud. Teemade käsitlus väga põhjalik, algtasemest kuni kaugele edasijõudnu tasemeni. Kolmest kõige auväärsem vanuselt (esimene trükk juba 1988). Aluseks Haskelli iidne versioon 1.3, kuid see on pisipuudus.

Keelekirjeldus

[4] Jones, Simon Peyton (ed.): *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

Veebis aadressilt `http://www.haskell.org/definition/` kättesaadav. Seal on üleval ka tõlge vene keelde.

Paljude aastate jooksul keele loojate vahelistes vaidlustes kujunenud ning täpsusele ja täielikkusele pretendeeriv Haskell 98 spetsifikatsioon.

Koduleht

[5] *Haskell*. `http://www.haskell.org`.

Vikipõhimõttel toimiv Haskelli koduleht.

Indeks