Sissejuhatus

Funktsionaalne programmeerimisparadigma

Paradigma järgi liigitub funktsionaalne (ingl *functional*) programmeerimine koos loogilise (ingl *logic*) programmeerimisega deklaratiivse programmeerimise alla, vastandudes imperatiivsele programmeerimisele.

Kui imperatiivses (ingl *imperative*) programmeerimises kirjeldab kood vahetult arvutusprotsessi käiku, siis deklaratiivne (ingl *declarative*) programmeerimine tähendab seda, et programmikood kirjeldab otseselt võttes matemaatilisi väärtusi ja abstraktseid seoseid nende vahel.

Loomulikult vastab deklaratiivses keeles kirjutatud korrektsele koodile ka mingi arvutusprotsess, mis koodi jooksutamisel toimub, sest jutt on ikkagi programmeerimiskeeltest. Kuid, erinevalt imperatiivsest programmeerimisest, pole arvutusprotsessi detailid deklaratiivses keeles kirjutatud koodis ilmutatud, need on määratud kaudselt.

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 muudab koodi hoolduse vähem töömahukaks. Modulaarsuse mitmeid vorme realiseerivad ka tänapäeva tuntud imperatiivsed programmeerimiskeeled.

Niisiis võimaldab deklaratiivne paradigma esitada abstraktsed seosed arvutusprotsessi osaliste — näiteks sisendi ja väljundi — vahel ilma protsessi detailideta. Nii saab deklaratiivses keeles kirjutatud koodi mõtet suurel määral mõista ka täiesti ilma tähelepanu pööramata sellele, kuidas masin selle järgi asju reaalselt arvutaks. Võrdluseks, imperatiivses keeles kirjutatud koodi mõte selgubki alles arvutusprotsessi detailidest.

Teisest küljest nõuab edukas programmeerimine deklaratiivses keeles rohkem õppimist ja mõttetööd kui imperatiivses, sest kuigi arvutusprotsessi detaile pole vaja kirjutada, tuleb nendega siiski arvestada ja nende ilmutamata olekul on see raskem. Programmeerija peab saama aru, kuidas tema kirjutatud koodi järgi arvutus toimub, kuigi seda koodis otseselt kirjas ei ole. Sama abstraktset funktsiooni on võimalik arvutada kiiremini või aeglasemalt ja pole kasu koodist, mis kirjeldab abstraktselt küll õiget seost, kuid mille järgi arvutamine võtab rohkem aega kui on Universumi

eluiga.

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

Funktsionaalse paradigma olemuslikud omadused

Käsitleme siin lähemalt tunnuseid, mis on omased eelkõige funktsionaalsele paradigmale.

Esimene tunnus on avaldiste väärtustamise keskne roll arvutusprotsessis. 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.

Teine tunnus on üksikandmete, andmestruktuuride, funktsioonide ja protseduuride süntaktiline eristamatus. Funktsioonid on andmed nagu näiteks täisarvudki ja funktsioonid saavad seega esineda kõigis neis rollides kus täisarvud. Näiteks võib iga funktsioon olla mõne avaldise väärtuseks, teise funktsiooni argumendiks või tulemuseks või mõne andmestruktuuri komponendiks. Selle kohta öeldakse, et funktsioonid on **esimese kategooria** (ingl *first-class*) väärtused; sama kehtib ka andmestruktuuride ja protseduuride kohta.

Puhtalt funktsionaalses keeles nagu Haskell, mida me hiljem lähemalt tundma saame, on tõepoolest ka protseduurid — arvutusprotsessid — käsitatud abstraktsete andmetena. On olemas funktsioonid, mis väiksematest protsessidest panevad kokku suuremaid. See võimaldabki soovitavat arvutusprotsessi esitada avaldise kujul. Avaldise "väärtustamine ja täitmine", millele ülal viitasime, tähendab avaldise väärtustamist ja väärtuseks oleva protsessi läbiviimist.

Kolmandana pöörame tähelepanu sellisele väga olulisele tunnusele nagu **ilmutatud viidatavus** (ingl *referential transparency*), mis sätestab, et avaldise mingi alamavaldise asendamisel temaga väärtuselt võrdsega avaldise väärtus ei muutu. Näiteks kui $\mathfrak{a}_1$ ja $\mathfrak{a}_2$ on avaldised, mille väärtuseks on mingi üks ja sama funktsioon, ning $\mathfrak{b}_1$ ja $\mathfrak{b}_2$ on avaldised, mille väärtused on võrdsed ja sobivad selle funktsiooni argumendiks, siis $\mathfrak{a}_1$ rakendamine $\mathfrak{b}_1$-le annab sama väärtuse nagu $\mathfrak{a}_2$ rakendamine $\mathfrak{b}_2$-le.

Ilmutatud viidatavus võib esmapilgul tunduda enesestmõistetav. Matemaatikas ta kehtib ja kannab Leibnizi seaduse nime ("võrdse asendamisel võrdsega saame võrdsed"). Samas imperatiivses programmeerimises ei saa ilmutatud viidatavuse kehtimisest juttugi olla. Imperatiivses keeles kirjutatud funktsioon $\mathfrak{a}$ võib väärtuse arvutamisel kasutada mõnd globaalset muutujat, mille väärtus $\mathfrak{a}$ kahe väljakutse vahepeal on muutunud. Seetõttu võib isegi sama $\mathfrak{a}$ rakendamisel samale avaldisele $\mathfrak{b}$ tulla välja kord üks, kord teine väärtus. Seda nimetatakse **kõrvalefektiks** (ingl *side-effect*), niisiis on ilmutatud viidatavus teisiti öeldes kõrvalefektide puudumine.

Siit saab selgeks, et ilmutatud viidatavusel on kolossaalsed järeldused programmeerimiskeele võimaluste kohta.

Näiteks ei saa ühegi muutuja väärtus kunagi sama ploki samal lugemisel muutuda. Muutujad pole muutujad samas mõttes mis imperatiivses programmeerimises. Funktsiooni erinevatel väljakutsetel võivad formaalsed parameetrid omandada küll erinevaid väärtusi, kuid funktsiooni väärtustamise 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 ($x + x = 2x$ on õige ainult tänu sellele, et valemi samal lugemisel on x igal pool ühe ja sama väärtusega).

Seega on muutujale omistamine funktsionaalses keeles võimatu. Samuti puuduvad imperatiivsetest keeltest tuttavad tsüklikonstruktsioonid. Kõik tsüklilised protsessid tuleb realiseerida rekursiooniga.

Isegi välise maailma sündmustel (klahvivajutused jms) ei ole muutujate väärtustele mingit mõju. Funktsionaalne keel tekitab jäiga piiri väärtuste maailma ning sündmuste maailma vahele. Alalises muutumises olev sündmuste maailm on "puhtast" väärtuste maailmast eraldatud nagu džinn pudelis. See on veel üks modulaarsuse vorm.

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 andmete, väärtuste, avaldiste jne klassifikatsiooni tüüpidesse koos tüüpide omaduste ja omavaheliste suhete võrgustikuga nimetatakse tüübisüsteemiks (ingl *type system*). 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 (ingl *lazy*) väärtustamine tähendab seda, et avaldisi väärtustatakse vastavalt vajadusele: kui mõni funktsioon oma väärtuse leidmisel argumenti ei kasuta, siis argumenti ei väärtustatagi, ja komplekssеid 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 (ingl *eager*) väärtustamine, mis tähendab, et funktsiooni väljakutsel kõigepealt väärtustatakse argument ja seda täielikult.

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. Ohjeldamatu laisk

väärtustamine ilma ilmutatud viidatavuseta muudaks programmide mõtte äärmiselt raskesti haaratavaks, sest funktsiooni argumendi väärtus sõltuks sellest, millisel hetkel ta välja arvutatakse.

Funktsionaalse paradigma tunnused on konkreetsed teed saavutamaks deklaratiivse programmeerimise põhieesmärki, mille järgi kood peab otseselt esitama abstraktseid seoseid matemaatiliste väärtuste vahel.

Head ja halvad küljed

Tuues välja funktsionaalse programmeerimise eeliseid, mainime kõigepealt koodi suurt väljendusvõimsust ja paindlikkust erinevate ülesannete lahendamisel. Tänu sellele, et funktsioonid on esimese kategooria väärtused 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 programmeerida ümberkäimist 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 üleminekul imperatiivselt keelelt funktsionaalsele mobiiltarkvara arendamisel kiirendas programmeerijate töökiirust isegi kuni kümme korda.

Ilmutatud viidatavus võimaldab koodi mõista matemaatikast tuttaval moel, see lihtsustab koodist arusaamist ja 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).

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 ka programmide kompileerimisel saadud masinkeelse programmi suuruse kohta — isegi triviaalne Haskell-programm võtab kompileeritult sadu kilobaite ruumi).

Programmeerimises näib kehtivat kuldreegel: mida vähem kulutame ressursse programmeerimisele, seda rohkem kulutame ressurssi programmide kasutamisel. Funktsionaalne keel võimaldab olulist säästu programmeerija tööajas, kuid funktsionaalses keeles kirjutatud program-

mid 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. Seoses protsessorite kiiruse pideva suurenemisega kasvab nende ülesannete hulk, mille puhul viimase peal kiirus pole enam vajalik, ja ühtlasi funktsionaalse programmeerimise läbilöögivõime.

Teine funktsionaalse programmeerimise puudus seondub programmeerimisega abstraktsel tasemel. Et see oleks võimalik, peab mäluhaldus (mälu reserveerimine ja vabastamine, prügikoristus) toimuma automaatselt. Programmeerijal pole võimalik endal mäluhaldust organiseerida. Funktsionaalne keel ei võimalda nii madala taseme asju väljendada. Kuid mõnikord oleks otsene mäluhaldamine 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 seal on niisuguste avaldiste kirjutamiseks välja töötatud spetsiaalne süntaks, mis võimaldab Haskellis interaktiivseid tegevusi programmeerida peaaegu niisama ladusalt kui mistahes imperatiivses keeles. Seega võib öelda, et siin on tegemist juba praktiliselt ületatud probleemiga.

Haskelli väärtuste maailm

Käesolevas õppematerjalis on funktsionaalse paradigma näidiskeelena kasutatud Haskelli. Enne aga, kui tutvume Haskelli süntaksiga ja jõuame esimeste funktsionaalsete programmide kirjutamiseni, anname lühiülevaate Haskellist tema semantika poole pealt.

Järgnevad jaotised visandavad pildi maailmast, milles Haskell-koodi tuleb mõista. Haskellis programmeerides tuleb mõelda üsnagi teistsugustes terminites võrreldes imperatiivsete keeltega, see maailm sarnaneb palju rohkem matemaatika struktuuridega. Samas tuleb tunda mitmeid mõisteid nende tähenduses, mis matemaatikas pole standardne, vaid on spetsiifiline just Haskellile (ja võibolla veel mõnele sarnasele programmeerimiskeelele).

Olles nende iseärasustega eelnevalt kursis, on süntaksi õppimine ja süntaktiliste objektide tähenduse tabamine hõlpsam.

Matemaatilisi objekte, millena Haskell-koodi osi interpreteerida, nimetame üldiselt **väärtusteks** (ingl *value*). Kõige jämedamas plaanis jagunevad Haskelli väärtused andmeteks, tüüpideks, liikideks ja klassideks. Tüüpidel, liikidel ja klassidel on klassifitseeriv funktsioon. Tüübid klassifitseerivad andmeid, liigid ja klassid omakorda tüüpe. Järgnev tutvustav ringkäik on üles

ehitatud nende klassifitseerimisvahekordade kaupa.

Andmed ja tüübid

Nagu juba funktsionaalse paradigma kirjelduses juttu oli, peetakse selles paradigmas väga erinevaid objekte **andmeteks** (ingl *data*). Haskellis teevad erinevat laadi andmete vahel vahet **tüübid** (ingl *type*). Tüübisüsteem on väga rikas.

Alustame lihtsamast. Tüüpi Int kuuluvad 4-baidised märgiga täisarvud. Vähim täisarv tüübis Int on -2147483648, suurim on 2147483647. On ka teine täisarvutüüp — Integer, mis arvudele suuruspiiranguid ei sea. Tüüp Integer on niisiis põhimõtteliselt lõpmatu. Tüübid Float ja Double on kaks erinevat ujukomaarvutüüpi. Väärtused 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 — suurusvahekorratüüp. Väärtused sellest tüübist tähistavad kahe võrreldava objekti võrdlemise tulemusi. Võimalused on “väiksem”, “niisama suur”, “suurem”, mida tähistame vastavalt LT, EQ, GT.

Reaalses elus ei pruugi arvutus lõpetada normaalse väärtuse väljastamisega. Hõlmamaks ka n-ö halbu juhte, loetakse iga tüüp sisaldavaks peale oma põhiväärtuste ka erilist väärtust $\perp$, mida nimetatakse **bottomiks** (ingl *bottom*). Bottom tähendab normaalsel kujul info täielikku puudumist väärtuses. Näiteks täisarvude 1 ja 0 jagamisel on väärtuseks $\perp$. Koos bottomiga sisaldab näiteks Bool kolme ja Ordering nelja väärtust.

Tüüpi nimetatakse **loetelutüübiks** (ingl *enumeration type*), kui ükski tema väärtus ei sisalda endas väärtust mõnest teisest tüübist. Teisiti öeldes, loetelutüübid on need, mis pole konstrueeritud mõne teise tüübi peale. Mõnes mõttes võib loetelutüüpi mõista ka kui sellist, mida pole võimalik lihtsamalt formaalselt defineerida kui loetledes kõik temasse kuuluvad väärtused. Kõik senivaadeldud tüübid on loetelutüübid.

Loetelutüüpidele vastanduvad struktuuritüübid, mille väärtusteks on **andmestruktuurid** (ingl *data structure*), kuid ka loetelutüübi väärtusi võib lugeda triviaalseteks andmestruktuurideks, mis kunagi midagi ei sisalda.

Näitena mittetriviaalsetest andmestruktuuritüüpidest vaatleme kõigepealt listitüüpe. List (ingl *list*) on üht tüüpi andmete kogum, kus andmed on struktureeritud ühte jorru. List on Haskellis ja üldse funktsionaalses programmeerimises põhilisimaks kasutatavaks andmestruktuuriks.

Lihtsaim list on **tühi** (ingl *empty*) — tähistagu seda kirjutis []. Mittetühjad listid avalduvad operatsiooni : abil, mis tähendab ühe andme lisamist listi algusse. Nii on täisarvude listid näiteks $1:[]$, $0:1:[]$, $13:31:[]$, $1:3:5:7:9:[]$ ja muidugi [] ise. Näiteks $\mathrm{True}:[]$, $\mathrm{False}:[]$,

$\mathrm{True} : \mathrm{False} : [], \mathrm{False} : \mathrm{False} : \mathrm{False} : \mathrm{False} : []$ — ja jälle muidugi [] ise — on tõeväärtuste listid.

Andmeid, mis on operatsiooniga : listi paigutatud, nimetatakse listi **elementideks** (ingl *element*). Kui a on anne ja l on a-ga sama tüüpi andmete list, siis $a : l$ on ka sama tüüpi andmete list, kus elementideks on kõik need andmed mis listis l ja lisaks a. Niisiis $a : b : l$ on tegelikult sama mis $a : (b : l)$. Listis $a : l$ nimetatakse elementi a listi **peaks** (ingl *head*) ja alamstruktuuri l listi **sabaks** (ingl *tail*). Kui l on mingi list, siis l **alamlistideks** (ingl *sublist*) nimetatakse listi l ja, kui ta on mittetühi, ka l saba kõiki alamliste.

Vastanduvalt eespool toodud näidetele listidest on oluline rõhutada, et Haskelli list ei pea olema lõplik. Näiteks on olemas **lõpmatud** (ingl *infinite*) listid $0 : 1 : 2 : 3 : \ldots$, $\mathrm{False} : \mathrm{False} : \mathrm{False} : \mathrm{False} : \ldots$ jms. Sellised listid 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** (ingl *partial*). Osalised listid on näiteks $1 : \bot$, $\mathrm{True} : \mathrm{False} : \mathrm{False} : \bot$ ja $\bot$ ise.

Laisk väärtustamine teeb lõpmatud ja osalised listid omaette väärtustena mõttekaks. Seejuures näiteks $\bot$ ja $1 : \bot$ on erinevad väärtused: 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.

Kõik listid jagunevad tüüpidesse vastavalt elementide tüübile; ei ole olemas üht listitüüpi, kuhu kuuluksid kõik listid. Listid elementidega tüübist Int moodustavad tüübi $\mathrm{List\,Int}$, sümbolitüüpi elementidega listid tüübi $\mathrm{List\,Char}$ jne, mistahes tüübi A korral listitüüpi, millesse kuuluvad listid elementidega tüübist A, tähistame $\mathrm{List}\ A$. Kuna A võib olla tõesti ükskõik milline tüüp, siis saame konstrueerida näiteks tüüpide jada $\mathrm{Int}, \mathrm{List\,Int}, \mathrm{List\,(List\,Int)}, \ldots$. See näitab, et Haskellis on tüüpe lõpmata palju.

String on Haskellis sama mis sümbolite list. Niisiis tüüp $\mathrm{List\,Char}$ on ühtlasi stringitüüp. Vastavus on järgmine: tühi string kujutab endast tühja listi; mittetühja stringi pea on tema esimene sümbol ning saba tema alamstring alates teisest sümbolist kuni lõpuni. Seega on Haskellis olemas lõpmatud stringid.

Kuigi Haskellis on igas tüübis olemas väärtus $\bot$, mis tähendab oodatava normaalse väärtuse puudumist, ei ole see alati piisav vahend potentsiaalselt ebaõnnestuva arvutusega ümberkäimisel. Bottomi alla kuuluvad ka kõige hullemad ebaõnnestumised, mida ei ole kunagi võimalik üldse kindlaks teha, nagu mõtlema jäämine lõpmata kauaks. Seepärast on kasulikud tüübid, mis võimaldavad infot oodatava väärtuse puudumise kohta esitada ka normaalse väärtuse kujul. Selleks on olemas tüübid $\mathrm{Maybe}\ A$.

Kui A on mingi tüüp, siis tüüp $\mathrm{Maybe}\ A$ sisaldab väärtustena kõiki A väärtusi a kujul $\mathrm{Just}\ a$ ja lisaks väärtust $\mathrm{Nothing}$. Näiteks tüüp $\mathrm{Maybe\,Int}$ sisaldab muuhulgas väärtusi $\mathrm{Just}\,0$, $\mathrm{Just}\,1$, $\mathrm{Just}\,(-18)$ ja $\mathrm{Nothing}$, tüüp $\mathrm{Maybe\,Bool}$ koosneb aga väärtustest $\mathrm{Just\,True}$, $\mathrm{Just\,False}$ ja $\mathrm{Nothing}$ (rohkem normaalseid väärtusi pole). Väärtus $\mathrm{Nothing}$ tähendab tavalise väärtuse puudumist, väärtus $\mathrm{Just}\ a$ aga seda, et oodatav A-tüüpi väärtus on a. Tüüpi $\mathrm{Maybe}\ A$ võib interpreteerida kui listitüüpi, kus list ei saa sisaldada üle ühe elemendi.

Palju kasutust leiavad **järjendid** (ingl *tuple*), mis on sarnased järjendite ehk vektoritega matemaatikas. Järjenditüüpide kohta öeldakse ka **korrutistüüp** (ingl *product type*), sest kui järjenditüübi bottom kõrvale jätta, siis on järjenditüübi näol matemaatiliselt tegemist otsekorrutisega. Korrutistüüpide alla kuuluvad paaritüübid, kolmikutüübid jne. Kui A ja B on tüübid, siis $A \times B$ on tüüp, mis sisaldab väärtuste **paarid** (ingl *pair*) $(a\,,b)$, kus a tüüp on A ja b tüüp on B. Samamoodi $A \times B \times C$ sisaldab **kolmikuid** (ingl *triple*) $(a\,,b\,,c)$, kus a tüüp on A, b tüüp B ja c tüüp C.

On olemas **ühiktüüp** (ingl *unit type*), milles on ainult üks normaalne väärtus, mida tähistame $()$. Ühiktüüpi võib pidada korrutistüübiks, kus tegurtüüpide arv on 0.

Korrutistüübile vastandub **summatüüp** (ingl *sum type*). Kui A ja B on tüübid, siis $\mathrm{Either}\ A\ B$ on tüüp, mis sisaldab väärtused $\mathrm{Left}\ a$ ja $\mathrm{Right}\ b$, kus a tüüp on A ja b tüüp on B. Summatüübi nimetus tuleb sellest, et kui bottomeid mitte arvestada, on väärtusi tüübis $\mathrm{Either}\ A\ B$ samapalju kui tüüpides A ja B kokku.

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

Näiteks paaritüübi bottomit võidakse Haskellis küll paariks nimetada, kuid matemaatilises mõttes ta paar ei ole. Nõudes 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 sisaldab see paaritüüp ka väärtust $(\bot\,,\bot)$, mis on paar nagu muiste, kuid annab sarnaselt $\bot$-ga komponentide pärimisel 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 andmestruktuuritüüpide korral. Teisalt võib puhta südamega märkida, et ainsad selle kriteeriumi kontranäited on põhimõtteliselt sellised, nagu eelmises lõigus kirjeldatud.

Andmestruktuuridest hoopis erinevat laadi väärtused on funktsioonid. **Funktsioon** (ingl *function*) saab mingi väärtuse **argumendiks** (ingl *argument*) ja arvutab selle põhjal mingi (üldjuhul uue) väärtuse. Argumendi andmist funktsioonile nimetatakse funktsiooni **rakendamiseks** (ingl *application*) argumendile. Kui f on funktsioon ja x talle sobiv argument, siis f rakendamise tulemust argumendile x (teisi sõnu, f väärtust kohal x) 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 koodiks, on tüüpi $\mathrm{Char} \to \mathrm{Int}$, siinus on tüüpi $\mathrm{Float} \to \mathrm{Float}$ või $\mathrm{Double} \to \mathrm{Double}$ (st on kaks siinusfunktsiooni, üks kummagi ujukomaarvutüübi jaoks) jne. Maybe-tüüpide juures tutvusime funktsioonidega Just, mis on tüüpi $A \to \mathrm{Maybe}\ A$ iga tüübi A jaoks.

Kuna funktsionaalses keeles võib iga funktsioon olla funktsiooni argument, samuti väärtus mingil argumendil, võivad funktsiooni argumenditüüp ja väärtusetüüp olla omakorda funktsioonitüübid. Näiteks tüüpi (Integer $\rightarrow$ Integer) $\rightarrow$ (Bool $\rightarrow$ Char) kuuluvad parajasti funktsioonid, mis võtavad argumendiks täisarve teisendavaid funktsioone ja annavad väärtuseks funktsioone, mis omakorda võtavad argumendiks tõeväärtusi ja annavad tulemuseks sümboleid. Kokkuleppeliselt võib paremalt sulud ära jätta: $A \rightarrow B \rightarrow C$ tähendab sama mis $A \rightarrow (B \rightarrow C)$. Seega selles lõigus näitena toodud funktsioonitüübi võib kirjutada ka ühe sulupaariga kujul (Integer $\rightarrow$ Integer) $\rightarrow$ Bool $\rightarrow$ Char.

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 $f(x, y) = \sin x \cdot \cos y$ saab kujutada funktsioonina, mille argumenditüüp on Double $\times$ Double ja väärtusetüüp Double; siis funktsioon ise on tüüpi Double $\times$ Double $\rightarrow$ Double. Teine võimalus on kasutada *curried*-kujulisi funktsioone.

Kui f on funktsioon, mille argumentideks on mingid järjendid, siis f ***curried*-kuju** curry f on funktsioon, mis töötab sisuliselt samamoodi nagu f, kuid võtab argumendid ükshaaval. Kui f tüüp on $A_1 \times \ldots \times A_l \rightarrow B$, siis curry f tüüp on $A_1 \rightarrow \ldots \rightarrow A_l \rightarrow B$ ning mistahes väärtuste $a_1, \ldots, a_l$ korral vastavalt tüüpidest $A_1, \ldots, A_l$ kehtib

$$f(a_1\,, \ldots\,, a_l) = \text{curry}\; f\; a_1 \ldots a_l.$$

Ülal vaatlesime funktsiooni f, mis võtab argumendiks ujukomaarvude paari $(x\,, y)$ ja annab sellel tulemuseks ujukomaarvu $\sin x \cdot \cos y$. Tema *curried*-kuju curry f on funktsioon tüüpi Double $\rightarrow$ Double $\rightarrow$ Double, mis võtab argumendiks ühe ujukomaarvu x ja annab tulemuseks funktsiooni, mis omakorda võtab argumendiks ujukomaarvu y ning annab tulemuseks ujukomaarvu $\sin x \cdot \cos y$.

Lugeja võis märgata, et ka curry on ise *curried*-kujuline funktsioon. Eelmises lõigus oli tema tüüp (Double $\times$ Double $\rightarrow$ Double) $\rightarrow$ Double $\rightarrow$ Double $\rightarrow$ Double, sest ta võttis argumendiks funktsiooni tüübist Double $\times$ Double $\rightarrow$ Double ja andis tulemuseks funktsiooni tüübist Double $\rightarrow$ Double $\rightarrow$ Double.

Curried-kujuline funktsioon on seega sama mis funktsioon, mille väärtusetüüp on funktsioonitüüp. Haskellis on *curried*-kujulised funktsioonid väga tavalised, enamik standardsetest mitmeparameetrilistest funktsionaalsetest suhetest (mh kõik aritmeetilised tehted) on realiseeritud *curried*-kujuliselt.

Haskellis (ja üldse funktsionaalses programmeerimises) on väga tavalised ka funktsioonid, mille argumenditüüp on funktsioonitüüp. Selliseid funktsioone nimetatakse **kõrgemat järku** (ingl *higher-order*) funktsioonideks. Järgu mõistet võib ka täpsustada, see käiks järgmiselt.

1. 0. järku funktsiooniks nimetatakse suvalist väärtust, mis pole funktsioon.

2. Kui n on naturaalarv, siis (n + 1). järku funktsiooniks nimetatakse suvalist funktsiooni, mille argumenditüüp sisaldab n. järku funktsioone.

Näiteks siinus tüübist Double $\rightarrow$ Double on 1. järku funktsioon, sest Double pole funktsioonitüüp; $f(x\,, y) = \sin x \cdot \cos y$ on 1. järku funktsioon, sest Double $\times$ Double pole funktsioonitüüp, ka curry f on 1. järku; curry ise on aga 2. järku funktsioon, sest tema argumenditüüp on 1. järku funktsioonide tüüp. Funktsioon on kõrgemat järku, kui tema järk on vähemalt 2.

Öeldakse, et funktsioon f on **agar** (ingl *strict*), kui $f \perp = \perp$. Vastasel korral ütleme, et f on **laisk**. Agara funktsiooni mõiste on agara väärtustamise peegeldus funktsioonidele väärtuste maailmas. 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 see väärtus on ka paratamatult bottom. Nagu võib arvata, tuleb Haskellis tihti ette funktsioone, mis pole agarad.

Funktsionaalse paradigma omadused tagavad, et Haskelli funktsioonide käsitlemine nii, nagu funktsioone ka 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. Kui matemaatikas kehtib ekstensionaalsus — mistahes funktsioonide f, g korral kui f ja g töötavad igal argumendil ühtmoodi, siis f ja g on võrdsed —, siis Haskellis on bottom eristatav funktsioonist, mis ise ei ole bottom, kuid annab igal argumendil tulemuseks bottomi.

Funktsioonide interpreteerimine matemaatika moodi eeldab ilmutatud viidatavust: samal argumendil annab sama funktsioon sõltumata kõigest sama tulemuse. Kuid reaalses elus on enamasti tarvis, et programm saaks oma töö jooksul uusi sisendeid lugeda ja oma tööd neist sõltuvalt teha. Selle realiseerimine funktsioonidega on mõneti keeruline, põhimõtteliselt peaks funktsioon võtma lisaks oma muudele parameetritele argumendiks keskkonnaseisu, see aga nõuaks keskkonnaseisu esitamist keeles, mis tähendaks palju lisatööd.

Haskellis on mindud teist teed ja loodud eraldi tüübid, millesse kuuluvad just **protseduurid** (ingl *action*), mis võivad keskkonnast sõltuda ja keskkonda mõjutada. (Need tüübid võivad olla teatud erilised funktsioonitüübid, kuid see on jäetud realisatsiooniküsimuseks ja programmeerija ei pea sellest midagi teadma.) Iga tüübi A jaoks on olemas selline tüüp IO A, kusjuures A on protseduuri poolt **edastatava** (ingl *return*) väärtuse tüüp. Mudel on nimelt selline, et iga protseduur saab kasutada seda infot, mida varasemad protseduurid on edastanud, ja edastab lõpuks ise mingi väärtuse, mida järgnevad protseduurid saavad kasutada. Kogu arvutust võib võrrelda konveieriga, mis koosneb protseduuridest, mis teevad mingeid operatsioone ja edastavad üksteisele järjest väärtusi. Edastamine on ainus võimalus kasutada ja teha kättesaadavaks keskkonnast loetud uut infot.

Edastamisele kuulubki tavaliselt uus keskkonnast saadud info. Näiteks protseduur, mis loeb standardsisendist ühe sümboli ja edastab selle, kuulub tüüpi IO Char. Protseduur, mis loeb standardsisendist sümboli, aga edastab tema koodi, on tüüpi IO Int.

Seevastu protseduuril, mis kirjutab midagi standardväljundisse, näiteks sümboli a, ei ole tingimata tarvis midagi sisulist edastada, sest see protseduur ei too sisse uut infot. Haskelli standardteegis edastavad sellised protseduurid tüüpiliselt väärtuse () ühiktüübist. Muidugi on võimalik ka selline protseduur, mis kirjutab sümboli standardväljundisse ja ka edastab selle sümboli, selline oleks tüüpi IO Char.

Sama protseduur võib erinevatel kordadel sõltuvalt keskkonnast loetud infole teha erinevaid asju ja edastada erinevaid väärtusi (väärtuse 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 (selle, mille kasutaja valis).

Põhimõtteliselt on võimalik kõik algoritmid realiseerida protseduuride kaudu, aga selline lähenemine on funktsionaalses keeles halb programmeerimisstiil. Funktsionaalse keele üks eesmärke on eraldada matemaatilise funktsioonina esituvad 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.

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. Niisiis tüüpide Int ja Integer elemendid on üksteisest erinevad täisarvud, kuigi me neid ühtmoodi tähistame, samamoodi on Float ja Double elemendid üksteisest erinevad ujukomaarvud. Muidugi erineb väärtus Just a väärtusest a, kuid ka väärtus Nothing tüüpidest Maybe A on kõikide tüüpide A korral unikaalne. Isegi $\perp$ on igal tüübil oma.

Tüübid ja liigid

Eelmises jaotises nähtud tüübindus toob peale andmeid klassifitseerivate tüüpide kaasa ka tüüpidel määratud funktsioone ehk tüübifunktsioone.

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 List töötab tüüpidel ja annab välja tüüpe, on ta põhimõtteliselt erinev kõigist funktsioonidest funktsioonitüüpides, sest nood võtavad argumendiks ja annavad tulemuseks andmeid.

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 *curried*-kujuline. Mõnes rakenduses on mõttekad isegi kõrgemat järku tüübifunktsioonid.

Kui jutuks on korraga tüübid ja tüübifunktsioonid, kasutatakse tavaliselt mõlema kohta sõna

“tüüp”. See annab sõnale “tüüp” 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.

Laiemalt mõistetud tüüpidel on oma klassifikatsioon, mille annavad **liigid** (ingl *kind*). 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 andmete maailmas töötavatest funktsioonidest; tema liik on $*$. Seevastu tüübifunktsioon on funktsioon, mis töötab tüüpide maailmas; tema liik on kõike muud kui $*$.

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

Tüübid ja klassid

On veel teinegi tüüpe klassifitseeriv süsteem — **klassid** (ingl *class*). Ühte klassi kuuluvad tüübid 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** (ingl *subclass*) ja teist esimese **ülemklassiks** (ingl *superclass*). Ühel klassil võib olla mitu alam- või ülemklassi.

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.

Näiteks on Haskellis eeldefineeritud klass Eq, millesse kuuluvad sellised tüübid, mille elementide võrdumist ja mittevõrdumist saab kontrollida. Sellesse klassi kuulub üsna palju tüüpe, 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 kuuluvad sellised tüübid, millel lisaks elementide 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; tõe-

väärtustüübil 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 siia kuulumine nõuaks kuulumist klassi Eq.

Klassi Show kuuluvad tüübid, mille elementidel on defineeritud stringiksteisendamine. Klassis Read aga on tüübid, mille elemente on võimalik stringikujust sisse lugeda. Enamik üldkasutatavaid tüüpe peale funktsiooni- ja protseduuritüüpide kuulub mõlemasse neist klassidest.

Klass Num sisaldab arvutüüpe. Kuuluvuskriteeriumiks on tüübis sisalduvatel väärtustel 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.

Klass Enum sisaldab tüüpe, mille elementide puhul saab rääkida järgmise ja eelmise leidmisest. Tüüpiliselt kuuluvad siia loetelutüübid. Klassis Bounded on sellised tüübid, millel on olemas vähim ja suurim element. Tuttavatest tüüpidest kuuluvad siia ühiktüüp, Int, Char, Bool ja Ordering. Klassi Random kuuluvad tüübid, millest on võimalik juhuarvude generaatoritega juhuslikult väärtusi valida. Siia kuuluvad vaikimisi olemasolevad täis- ja ujukomaarvutüübid, Char ja Bool.

Tüübiklassindus sarnaneb äratuntavalt objektorienteeritud programmeerimise klassisüsteemiga. Ka terminoloogia on siin sarnane, näiteks nimetatakse muutujaid, mis kannavad klassikuuluvuseks vajalikke operatsioone, meetoditeks. Oluline vahe on selles, et Haskelli klasside esindajad on tüübid, objektorienteeritud programmeerimises aga objektid ehk tavalised väärtused. Objektorienteeritud programmeerimises täidavad klassid tüüpide aset, Haskellis asuvad klassid järgu võrra kõrgemal.

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, komponentide 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. (Juhime jällegi tähelepanu, 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.

Esmatutvus töövahendite ja süntaksiga

Töö Haskelliga

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. Koodipaigutuse põhimõtete ja moodulisüsteemi kohta saab hõlpsasti lisainfot leida Haskelli tutvustavatest allikatest.

Haskelli töövahendid

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 testida koodi “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 vastab süsteem avaldise väärtusega, mille ta väljastamiseks mingil moel stringiks teisendab.

Kõigi Haskelli realisatsioonidega tuleb kaasa süsteemne moodulite teek, kus palju asju on eeldefineeritud. Teegis on kesksel kohal moodul Prelude, mis sisaldab põhilisimat vajalikku, muuhulgas näiteks aritmeetikat. Iga interaktiivse keskkonna käivitumisel loetakse moodul Prelude automaatselt sisse, teistes teegi moodulites defineeritud ning omadefineeritud nimede kasutamiseks tuleb eelnevalt anda mooduli või koodifaili sisselugemise käsk.

Vead koodis on erineva iseloomuga. Mõned, peamiselt süntaksi- ja tüübivead, avastab iga Haskelli realisatsioon juba koodifaili sisselugemisel ilma koodi jooksutamata; neid nimetatakse staatilisteks vigadeks. Nende esinemisel pole koodi kasutamine enne nende parandamist võimalik.

Osa vigu ilmneb muidugi alles koodi jooksutamisel, need on täitmisaegsed (ingl *runtime*) vead.

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

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

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

Terminali käsurealt käivitub Hugs käsuga `hugs`. Seejärel on ta kohe valmis vastu võtma Haskell-avaldisi ja neile reageerima. Kuna Haskelli süntaks on matemaatikalähedane, on korrektsete avaldiste sisestamine ja niiviisi interaktiivse keskkonna kasutamine taskuarvuti 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.

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

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 `:e` hea selle poolest, et redaktorist väljumisel sooritab Hugs kohe automaatselt ka staatiliste vigade kontrolli.

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 kuhjamälu suurusega 32 megabaiti ning `+kls` seab teatavad kolm eelistust, mille tähendust võib uurida Hugsis käsuga `:s` ilma argumentideta.

`:q`	Väljumine.
`:l`	Mooduli sisselugemine — argumendiks faili või mooduli nimi.
`:r`	Viimase sisselugemiskäsu kordamine.
`:t`	Avaldise tüübi kuvamine — argumendiks avaldis.
`:i`	Info nimede kohta — argumendiks nimede loend.
`:b`	Info moodulite eksporditavate nimede kohta — argumendiks moodulite loend.
`:e`	Tekstiredaktori avamine oma mooduli kirjutamiseks — argumendiks faili nimi.
`:?`	Hugsi käskude nimekirja kuvamine.
`:s`	Eelistuste ja parameetrite nimekirja kuvamine — ilma argumendita; eelistuste ja parameetrite muutmine — argumendiks muutuse kood, mis on kirjeldatud ilma argumendita saadavas nimekirjas.

Joonis 1: Peamised Hugsi käsud.

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.

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/`.

Kompilaatoriga 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 terminali 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.

GHC võimaldab Haskell-koodi kompileerida masinakoodiks, mida saab iseseisvalt käivitada. Selleks peab käivitatav protseduur olema failis defineeritud muutuja `main` väärtuseks.

Teeme näiteks programmi, mis kirjutab ekraanile avaldise `2 + (-3)` väärtuse ja mis asub failis nimega `Hello.hs`. Selle avaldise väärtus on arv, mitte protseduur; ekraanile kirjutamiseks tuleb talle rakendada operaator `print`. Sellega on kõik soovitud operatsioonid kirjeldatud ning esimene Haskellis kirjutatud programm oleks

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

Programmi p kompileerimiseks GHC-ga masinakoodiks võib anda käsu

`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.

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

`nhc98 -o` q p.

NHC-ga tuleb kaasa kompileerimismanageerija 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. Kuna kompileerimine toimub jooksvalt, kulub selleks iga avaldise väärtustamise eel tuntav ajavahemik; kui aga avaldise väärtustamine on mahukas, teeb kompileeritud koodi kiirus selle kaotuse kuhjaga tasa.

Koodifaili struktuur

Haskell võimaldab koodi kirjutada ilma selliste tülikate eraldajateta nagu on semikoolonid paljudes keeltes, aga selleks peavad koodi osad olema failis üksteise suhtes normaalselt rajastatud.

Normaalne rajastamine ei tähenda midagi väga konkreetset. Väga erineva maitse järgi rajastamine on lubatud. 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 õppevahendis on järgitud üht võimalikku paigutussüsteemi, mida täpselt jäljendades ei tohiks rajastusvigu tekkida. Igaüks võib leida katseliselt omale meelepäraseima rajastusviisi, mida ka Haskell aktsepteerib.

Haskell-koodi on siiski võimalik struktureerida ka semikooloni abil. Parsimisel Haskell lisabki struktureerimata koodile tema struktuuri ilmutatult näitavad semikoolonid. See kajastub tihti veateadetes, mis viitavad ootamatule semikoolonile mingis reas, kus pole ühtki semikoolonit. Sellisel juhul tuleb aru saada, et viga on rajastamises.

Haskell-koodi saab kommentaare lisada kahel viisil. Pikkusest sõltumata on võimalik kommentaare paigutada kommentaarisulgude **{-** ja **-}** vahele. Kommentaar võib asuda ka teise kommentaari sees. Lühemad kommentaarid võib aga panna kahekordse sidekriipsu järele, selle kommentaari lõpetab reavahetus. Sellise kommentaari puhul on üldjuhul vaja kriipsude ja kommentaari alguse vahele eraldavat tühisümbolit, kuid kui kommentaar algab tähega, siis võib teda alustada otse sidekriipsude järelt.

Joonisel 2 esitatud sisuga näitekoodis on varem kirjutatud esimese Haskell-programmi koodile (1) lisatud hulk kommentaare.

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

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

Joonis 2: Kommentaaride kirjutamist illustreeriv näidisfail.

Ülesandeid

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

Haskell-koodifailide standardsed nimelaiendid on `.hs` ja `.lhs`, mis tähendavad vastavalt "*Haskell script*" ja "*literate Haskell script*". 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.)

Laiendiga `.lhs` failides loetakse koodiridadeks ainult need read, mis algavad sümboliga `>`, kood moodustub nende ridade algussümbolile järgnevatest osadest. Muudel ridadel võib lisada kommentaare või hoida millist tahes koodiga seonduvat infot. Süntaksireeglid `.lhs`-faili koodiosa kohta on samad mis `.hs`-faili puhul terve faili kohta.

Ülesandeid

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

Haskelli moodulid

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`.

Moodulid teenivad Haskellis ühelt poolt väärtuste temaatilise jagamise eesmärki: ratsionaalarvudega seonduvaid operatsioone saab moodulist Data.Ratio, kompleksarvudega seonduvaid moodulist Data.Complex jne. See on vaade mooduli kasutaja aspektist.

Haskellis on näiteks võimalik teha täpseid arvutusi harilike murdudega vastanduvalt ujukomaarvudele, millega opereerimine on ebatäpne. Kuid murrujoone aset täidab protsendimärk %, mida pole loetud põhiliste operaatorite hulka ja mida seetõttu moodulist Prelude ei saa. Selle sümboli kasutamiseks tuleb sisse lugeda moodul Data.Ratio. Teinud seda interaktiivses keskkonnas, saame anda arvutile ülesandeks teha tehteid nagu näiteks `1 % 2 + 1 % 3`, millele saame täpse vastuse `5 % 6`.

Mooduli kirjutaja aspektist teenivad moodulid kõigepealt koodi struktureerimise eesmärki, mis on mõneti erinev. Mitte kõik, mis teatavas moodulis defineeritakse, ei pea jääma selle mooduli kaudu kasutajale kättesaadavaks. Väljaspoolt on kättesaadavad ainult nimed, mida moodul **eksspordib** (ingl *export*). Teisest küljest ei pea kõik, mis mooduli kaudu kasutajale kättesaadav on, olema just selles moodulis defineeritud. Moodul võib ka teiste moodulite eksporditavaid nimesid **importida** (ingl *import*) ja omakorda edasi eksportida samadel alustel moodulis endas defineeritud nimedega.

Vaadeldud kahes aspektis on Haskelli moodul Java klassi analoog. Samas kui objektorienteeritud keelte reeglid nõuavad samalaadsete objektidega seonduvate isendimeetodite koodi 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. On täielikult programmeerija vastutusel, et kood moodulite vahel tõepoolest mõistlikult oleks jaotatud. See on tarkvara hoolduse aspekt.

Esimese tutvuse moodulitega nende programmeerija aspektist võib teha Hugsi teegi peal, sest Hugsi distributsioonis on teegi moodulid algteksti kujul. GHC ja NHC distributsioonis on teek vaid kompileeritud kujul, algtekste pole. Kõigi teekide algtekstid on siiski valdavalt samad, kuna pärinevad samast repositooriumist.

Ülesandeid

12. Visata silm peale Hugsi moodulite teegile.
13. Lugeda interaktiivses keskkonnas sisse mõni uus moodul ja küsida selle mooduli eksporditavate nimede loend.

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.

```
import Data.Ratio

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

Joonis 3: Programm impordideklaratsiooniga.

Programmi moodul, mis defineerib selle muutuja `main`, mille väärtuseks on käivitatav protseduur, peab olema nimega `Main`. Kui mooduli päis puudub, siis interpreteeritakse moodulit nii, nagu oleks tal päis **`module`** `Main` **`where`**, seega ilma päiseta mooduli nimi on `Main`.

Meie moodul failis `Hello.hs` (joonis 2) 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 otse sisse loevad (muidu pidanuks meie näitemoodul olema failis `Main.hs`).

Ülesandeid

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

Moodulis saab kasutada nimesid oma tähenduses, mis defineeritakse samas moodulis või imporditakse mõnest teisest. Peamooduli 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` $\mathtt{m}$,

kus $\mathtt{m}$ on mooduli nimi.

Näiteks kui tahame kirjutada harilikke murde oma moodulis, peame seal importima mooduli Data.Ratio. Joonisel 3 on esitatud programm, mis sooritab tehte $\frac{1}{2}+\frac{1}{3}$ ja kirjutab vastuse ekraanile.

Importimisel võib tekkida nimekollisioon, kui mõni nimi, mis moodulis defineeritakse, tuleb ka impordi kaudu sisse või imporditakse mõni nimi mitmest moodulist. Niikaua kui nime algpäritolu on üheselt tuvastatav, ei ole nime kasutamisel vaja täiendavalt mooduli nime lisada ehk

nime adresseerida (ingl *qualify*). Kuid see on alati lubatud, näiteks võib `%` asemel kirjutada `Data.Ratio.%` (mõned interaktiivsed keskkonnad küll oma käsureal seda ei luba).

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. Seda saab muuta, esitades mooduli päises mooduli nime järel sulgudes täpsustuse. Seejuures garanteeritakse, et iga 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 nimekollisiooni.

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

Ülesandeid

15. Kirjutada moodul, mille sisselugemisel interaktiivses interpretaatoris on adresseerimata kasutatavad nii ratsionaalarvud kui kompleksarvud.

Lihtsad avaldised

Avaldised (ingl *expression*), mida saab anda interaktiivsele keskkonnale väärtustamiseks või täitmiseks, pole ainus süntaktilise objekti liik Haskellis. Iseseisvalt käivitatava programmi tegemisel juba koostasime avaldisest erineva süntaktilise objekti — deklaratsiooni —, mis defineeris teatava protseduuri muutuja `main` väärtuseks.

Alustame oma ülevaadet Haskelli süntaksist siiski just avaldistega kahel põhjusel. Esiteks, neid saab kiiresti testida interaktiivse keskkonna käsurealt ilma programme kirjutamata. Teiseks, avaldised, kuigi nad pole kõige lihtsamad süntaktilised objektid Haskellis, on lugejale neist kõige tuttavamad, sest avaldisi kasutatakse üldjoontes samal kujul ka matemaatikas.

Vastavalt väärtusele tehakse vahet andmeavaldisel ja tüübiavaldisel: andmeavaldise väärtus on anne, tüübiavaldise väärtus on tüüp. Enamasti on asi kontekstist selge ja räägime lihtsalt avaldistest. Interaktiivse interpretaatori käsurealt saab väärtustada vaid andmeavaldisi.

Avaldist nimetatakse monomorfseks (ingl *monomorphic*), kui tema väärtuse tüüp on üheselt määratud, ja polümorfseks (ingl *polymorphic*), kui tal on väärtusi mitmest tüübist. Näiteks aritmeetilised avaldised on enamasti polümorfsed, sest arvud võivad kuuluda paljudesse eri tüüpidesse (Int, Integer, Double jne).

Kuna tihti on tüüpide käsitlemisel mugav ja tarvilik seostada andmeavaldisi mitte otseselt tüüpidega, mida võib olla üks või palju, vaid tüübiavaldisega, mis neid tüüpe väljendab, siis mõistetakse avaldise `a` tüübi all sellist tüübiavaldist, mis võtab kokku `a` kõigi väärtuste tüübid.

Haskellis on väga palju avaldiste alaliike, ühed lihtsamad, teised keerulisemad. Siin ja järgmises jaotises tegeleme lihtsate avaldistega, mille struktuuris ei leidu muid komplekssseid süntaktilisi objekte peale avaldiste ega ilmutatud hargnemist, ja teeme tutvust nende koostisosadega.

Märgime, et järgnevas on pidevalt paralleelselt vaatluse all ühelt poolt süntaktiline maailm, teisalt semantiline väärtuste maailm ning kolmandalt poolt ka arvutusprotsessid. Jõudumööda on neid ilmutatult üksteisest eristatud, kuid kuna need maailmad peegelduvad üksteises, siis on ühtesid ja samu termineid kasutatud erinevate maailmade objektide märkimiseks.

Näiteks on väärtuste maailmas paarid ja ka süntaktilises maailmas on paarid (formaalsed paarikujul kirjutised). Väärtuste maailmas räägime funktsioonide rakendamisest, kuid süntaktilisi konstruktsioone, mille tähenduseks on funktsiooni rakendamine, nimetame samuti rakendamiseks ning seejuures saab veel vahet teha rakendamisel kui operaatoril ja rakendamisel kui operaatori rakendamisega konstrueeritud avaldisel. Lõpuks nimetame **rakendamiseks** ka operaatori argumentidele rakendamisega konstrueeritud avaldise väärtustamisprotsessi.

Muutujad ja konstruktorid

Haskelli lihtsaimad tähendust kandvad süntaktilised objektid on **muutujad** (ingl *variable*) ja **konstruktorid** (ingl *constructor*). Analoogselt avaldiste jaotamisele jaotuvad muutujad väärtuse järgi **andmemuutujateks** ja **tüübimuutujateks**, konstruktorid aga **andmekonstruktoriteks** ja **tüübikonstruktoriteks**. Muutujad ja konstruktorid vastanduvad üksteisele selle poolest, kuidas arvutusprotsessi käigus nendega opereeritakse.

Muutuja tähistab Haskell-programmi täitmisel täiesti tundmatut väärtust. Paljas muutuja ei saa olla ühegi avaldise väärtuseks ega anda selle väärtuse kohta isegi mitte osalist informatsiooni. Kui arvutusprotsessis nõuab olukord muutuja väärtust, siis üritab Haskell teda väärtustada.

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

Muutujal võib normaalset väärtust ka mitte olla. Näiteks muutuja `undefined` väärtustamine ei anna välja muud kui veateate. Kui sõltumata keskkonna seisundist (kasutatav mälumaht, signaalid jms) ei õnnestu avaldise väärtustamise käigus leida tema väärtuse normaalset kuju, siis loetakse teoorias avaldise väärtuseks $\bot$. Normaalne kuju võib puududa kahel põhjusel: kas väärtustamine lõpeb veaga või ei lõpe üldse.

Niisiis muutuja `undefined` väärtus on $\bot$. Pangem tähele, et tegemist on defineeritud muutujaga, muidu tekiks süntaksiviga, mitte täitmisaegne viga.

Seevastu konstruktor tähistab Haskell-programmi täitmisel lõppväärtust, mida enam teisendada pole vaja. Konstruktor loetakse täisinformatsiooniks. Sellised on näiteks numbritega kirjutatud arvulised konstandid `0`, `1`, `1.5` jne, tõeväärtused `True` ja `False` ning tühja listi märkiv `[]`. Sama kehtib sümbolkonstantide kohta. Need kirjutatakse Haskellis sümbolina apostroofide vahel (nt `'A'`, `'!'`), kusjuures sümbolit võib asendada ka tema langjoonega algav kood. Nii on

korrektne sümbolkonstant näiteks `'\n'` (`\n` kodeerib Unixi reavahetust), ka teised C-st tuntud koodid on lubatud. Lisaks saab Haskellis kodeerida sümboleid ka kujul `\`s, kus s on sümboli ASCII-koodi kümnendesitus, ja kujul `\x`s, kus s on koodi kuueteistkümnendesitus. Nii tähendavad näiteks `\97` ja `\x61` sümbolit `a`. Langjoon sümbolina tuleb alati asendada koodiga, lihtsaim võimalus on kujul `\\`. Sümbolkonstandis on ka apostroofi kodeerimine kohustuslik, apostroofi saab kodeerida kujul `\'`.

Kui interaktiivse keskkonna käsurealt sisestada mõni eelpooltoodud konstruktor, tuleb vastuseks põhimõtteliselt seesama asi. Tõsi, mõnikord näeb see vastus teisiti välja, kuna vahepeal on toimunud sisendi parsimine ja stringiksteisendamine, mis võivad tulemuseks anda sisestatust erineva stringikuju.

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 koosnemine sümbolitest nimekirjas

`+`, `-`, `*`, `/`, `^`, `=`, `<`, `>`, `&`, `|`, `$`, `:`, `!`, `?`, `.`, `%`, `\`, `@`, `~`, `#`, (2)

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 (2), mispuhul ta peab algama kooloniga. Eeldefineeritud konstruktorite hulgas on ka mõned erandlikud, mille nimi ei vasta kirjeldatud tingimustele. Reeglipärased konstruktorid on ülalvaadelduist `True` ja `False`, ülejäänud on sisseehitatud erandlikud.

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

Ülesandeid

16. Anda interaktiivse interpretaatori käsurealt avaldis, mis sisaldab (a) väiketähega algavat defineerimata identifikaatorit, (b) suurtähega algavat defineerimata identifikaatorit. Lugeda mõlemal korral veateadet ja võrrelda veateatega, mis tuleb muutuja `undefined` väärtustamisel. Mida väljendavad erinevused?

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

Kõik senitoodud näited olid andmemuutujatest ja -konstruktoritest. Et tutvuda ka tüübimuutujate ja -konstruktoritega, vaatleme siinkohal avaldiste annoteerimist tüüpidega.

Annoteeritud avaldis on kujul

a `::` t, (3)

kus $\mathfrak{a}$ on andmeavaldis ja $\mathfrak{t}$ tüübiavaldis. Näiteks `True :: Bool` ja `'a' :: Char` on annoteeritud avaldised. Siin `Bool` ja `Char` on tüübikonstruktorid, mis tähendavad vastavalt tõeväärtustüüpi ja sümbolitüüpi.

Andes need annoteeritud avaldised interaktiivse keskkonna käsurealt väärtustada, saame vastuseks vastavalt `True` ja `'a'`. Üldiselt ongi tüübikorrektne annoteeritud avaldis väärtuselt võrdne annoteerimata avaldisega.

Kujul (3) oleva avaldise tüübikorrektsuse jaoks on tarvilik, et $\mathfrak{t}$ väljendaks ainult selliseid tüüpe, millesse kuuluvaid väärtusi saab $\mathfrak{a}$ omada. Eelnevad näited seda tingimust ka rahuldasid, kuna tõese tüüp on Bool ja sümboli a tüüp Char.

Polümorfse avaldise tüübi väljendamiseks võetakse appi tüübimuutujad. Näiteks muutuja `undefined` on polümorfne, tema väärtus on $\perp$ suvalisest tüübist. See tähendab, et `undefined` tüüp on `a`; siin `a` on lokaalne tüübimuutuja ja ta märgib suvalist tüüpi. Lokaalsete tüübimuutujate nimed peavad algama väiketähe või alakriipsuga.

Osa polümorfsete avaldiste tüüpe nõuab tüübimuutujatele klassikitsendusi. Näiteks arvkonstandi `1` tüüp on `(Num a) => a`; see tüübiavaldis väljendab suvalist tüüpi A, mille korral A kuulub klassi Num. Järelikult `1` omab väärtust parajasti igas tüübis A klassist Num ehk igas arvulises tüübis. Tingimust esitavat osa `(Num a)` nimetatakse **tüübikontekstiks**. Üldjuhul kujutab ta endast komadega eraldatud klassikuuluvustingimuste nimekirja ümarsulgudes.

Annoteerimise mõte on üldjuhul polümorfse avaldise väärtuse võimalike tüüpide hulka kitsendada. Kuigi me võime kirjutada `True :: Bool` või `1 :: (Num a) => a`, pole sellest mingit kasu, kuna süsteem teab niisamagi, et `True` tüüp on `Bool` ja `1` tüüp on `(Num a) => a`. Me võime aga kirjutada annoteeritud avaldise `1 :: Int`. Selle avaldise väärtus on 1 nimelt tüübist Int. Korrektsed on ka kirjutised `1 :: Integer`, `1 :: Float`, `1 :: Double`. Täiesti mõttekas on ka `1 :: (Floating a) => a`. Selle avaldise väärtus saab olla 1 suvalisest ujukomaarvulisest tüübist.

Kui polümorfse avaldise kontekstist ei selgu, millisesse tüüpi kuuluvat väärtust on mõeldud, võib süsteemil tekkida raskusi avaldise väärtustamisel. Osal juhtudel loetakse selline olukord isegi tüübiveaks ja väärtustamist ei alustatagi. Arvutüüpide puhul aga nii siiski ei juhtu, vaid süsteem valib võimalike tüüpide seast välja ühe. Vaikimisi loetakse avaldised, mis saavad omada kõiki täisarvutüüpe, tüüpi `Integer` ning avaldised, mis saavad omada kõiki ujukomaarvutüüpe, kuid mitte täisarvutüüpe, tüüpi `Double`.

Näiteks lastes interaktiivses keskkonnas väärtustada avaldise `1`, saame vastuseks `1`, sest süsteem loeb ta vaikimisi täisarvuks (täpsemalt tüüpi `Integer`). Kirjutades `1 :: (Floating a) => a`, tuleb aga vastuseks `1.0`, sest tüübiannotatsioon välistab täisarvulise interpretatsiooni (nüüd loetakse tüübiks `Double`).

Ülesandeid

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

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

Infiksoperaatorid

Muutujat või konstruktorit nimetatakse **infiksoperaatoriks** (ingl *infix operator*), kui teda kasutatakse infiksselt, st oma argumentide vahel. Interaktiivsete interpretaatoritega tutvumisel tõenäoliselt proovisite juba väärtustada aritmeetilisi avaldisi nagu nt `2 + 3` ja `2 * pi / 3`, kus avaldised on konstrueeritud atomaarsetest avaldistest infiksoperaatorite `+`, `-`, `*`, `/` abil.

Infiksoperaatorite ring on Haskellis väga lai. Aritmeetikagi ei piirdu mainitud nelja tehtemärgiga, jagamisest on olemas veel teinegi variant `%`. See infiksoperaator on kättesaadav moodulist Data.Ratio ja tema tähenduseks on põhimõtteliselt murrujoon: kahel täisarvul annab ta tulemuseks nende suhte ratsionaalarvuna. Ratsionaalarvude lõppkuju, millele nad väärtustamisel viiakse, defineeritakse kui esitus taandumatu murruna.

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 `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 väärtusteks on vastavalt konjunktsioon ja disjunktsioon. Nendega moodustatud loogilise avaldise väärtustamist alustatakse vasakust argumendist. Kui `&&` vasak argument on väär, siis paremat argumenti ei väärtustata. Kui `||` vasak argument on tõene, siis paremat ei väärtustata.

Iga infiksoperaatoriga seonduvad tema **prioriteet** (ingl *priority*) ja **assotsiatiivsus** (ingl *associativity*), mida on vältimatult vaja arvestada infiksoperaatorite vähegi keerukamal kasutamisel. Need atribuudid määravad avaldise struktuuri kohtades, kus sulud seda ei tee.

Prioriteete märgivad Haskellis täisarvud 0-st 9-ni, suurem arv vastab kõrgemale prioriteedile. Kohtades, kus sulud alamavaldiste järjekorda ei määra, tuleb kõrgema prioriteediga tehted teha varem. Teisi sõnu, kõrgema prioriteediga infiksoperaatorid seovad lekseeme 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 prioritee-

diga 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.

Kui sulud järjestust ei määra ja ka prioriteet on võistlevail infiksoperaatoritel võrdne, siis määrab järjestuse assotsiatiivsus. Infiksoperaator saab olla vasak-, parem- või mitteassotsiatiivne. Vasakassotsiatiivse (ingl *left associative*) operaatori esinemisi tuleb rakendada vasakult paremale, paremassotsiatiivse (ingl *right associative*) operaatori esinemisi paremalt vasakule. Näiteks astendamisoperaatorid on paremassotsiatiivsed, seega avaldis `9 ^ 9 ^ 9` on ekvivalentne avaldisega `9 ^ (9 ^ 9)`, mitte avaldisega `(9 ^ 9) ^ 9`.

Seevastu `-` ja `/` on sarnaselt matemaatikale vasakassotsiatiivsed. Assotsiatiivsus on oluline ka nendel operaatoritel, mille puhul lõppväärtus sooritamise järjekorrast ei sõltu, sest süsteem peab ju mingis järjekorras need tehted sooritama. Nii on määratud ka `+` ja `*` vasakassotsiatiivseks.

Mitteassotsiatiivsete (ingl *non-associative*) operatsioonide ahelesinemise puhul on järjekorra näitamine sulgudega kohustuslik, vastasel korral on tegemist süntaksiveaga. Sama kehtib juhul, kui ahelas esinevad vastandliku assotsiatiivsusega infiksoperaatorid.

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

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

Ülesandeid

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

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

22. Arvutada interaktiivses interpretaatoris arvu 1.6^{16} esitus taandumatu murruna.

Kõik seni vaadeldud infiksoperaatorid on muutujad. Ka konstruktorid saavad olla infiksoperaatorid. Sellisel juhul on tüüpiliselt tegemist andmestruktuure koostavate operatsioonidega.

Lihtsaimaks näiteks võiks tuua paaride moodustamise. Avaldise `(2 , -3)` väärtuseks on paar $(2, -3)$. Siin asub `,` argumentide `2` ja `-3` vahel. Paar võib olla ka keerulisemate komponentidega, nt `(1 + 1 , 2 - 5)`, mille väärtus on samuti $(2, -3)$.

Kui avaldis on paarikujul, siis tema väärtustusprotsess võib teisendada paari komponente, kuid sulud ja koma arvutuse käigus ära kaduda ei saa — paar jääb paariks. See näitab, et 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 kooloniga.

Ülesandeid

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

Teise näitena, seekord reeglipärasest konstruktorist, toome listikonstruktori `:`. Selle konstruktori väärtus on funktsioon, mis, saades argumendiks elemendi ja listi, annab välja uue listi, mille saab, kui lisab antud elemendi antud listi ette esimeseks elemendiks. Niisiis `:` eraldab mittetühja listi puhul tema pea ja saba, nii nagu `,` eraldab paari esimese ja teise komponendi. Kuna list Haskellis on üht tüüpi objektide ahelandmestruktuur, siis `:` vasak argument peab olema sama tüüpi mis parema argumendina saadud listi elemendid, muidu tekib tüübiviga.

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 seda avaldist kirjutada lihtsamalt `1 : 3 : 5 : 7 : 9 : []`.

Nagu paarigi korral, ei saa operaatoriga `:` moodustatud avaldisest teisendamise käigus see operaator kuhugi kaduda ega paigast nihkuda. Kui avaldis on kujul `a : b`, siis `:` jääb paika ja teisendatakse ainult avaldisi `a` ja `b`. See tähendab, et infiksoperaator `:` on konstruktor.

Konstruktori `:` rakendamisel mingile elemendile x ja listile l toimub ainult uue struktuuri lisamine; x ja l jäävad samale kujule nagu nad argumendiks tulid. Kui nad olid väärtustamata, siis nad jäävad väärtustamata, mis tähendab, et konstruktor `:` on laisk ja tema rakendamine käib konstantse keerukusega. Haskellis on enamik eeldefineeritud konstruktorfunktsioone sellised.

Interaktiivses keskkonnas listidega töötamisel võib algul segadust põhjustada asjaolu, et need keskkonnad esitavad liste kasutajale elementide loendina kantsulgudes. See ei tähenda, et arvutatud väärtusest need koolonid ja tühi list puuduvad, vastupidi. Aga interaktiivses keskkonnas lisandub väärtuse leidmisele tema stringina esitamise samm, mis võib väärtuse originaalstruktuuri moonutada.

Ülesandeid

24. Sisestada interaktiivses interpretaatoris mõned listid, jälgida, mida ta vastab, ja saada aru põhimõttest.

25. Selgitada välja operaatori `:` prioriteet senituntud infiksoperaatorite suhtes.

26. Lasta interpretaatori käsurealt väärtustada 3-elemendiline list, mille elemendid on baitide arvud vastavalt kilobaidis, megabaidis ja gigabaidis. Kasutada võimalikult vähe sulge.

27. Lasta interpretaatori käsurealt väärtustada 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.

28. Sisestada interpretaatori käsurealt list, mille elemendid on omakorda listid, millest vähemalt üks on tühi ja üks mittetühi. Kasutada võimalikult vähe sulge.

Paarikonstruktori ja listikonstruktori näidete baasil peaks argumente võtva konstruktori mõte olema arusaadav. Selliseid konstruktoreid võib avastada isegi Haskelli arvude maailmas. Standardteegi moodul Data.Complex annab kompleksarvude esituse kujul `a :+ b`, kus `a` väljendab reaalosa ja `b` imaginaarühiku i kordajat imaginaarosas ning need mõlemad peavad olema ujukomaarvulist tüüpi. Infiksoperaator `:+` on konstruktor. See muidugi tähendab, et Haskellis kujutatab kompleksarv endast paarilaadset andmestruktuuri. Mõneti erandlikult on konstruktor `:+` agar. Sellise korralduse põhjuseks on asjaolu, et enamasti pole mõtet hoida mälus kompleksarve, mis oleksid osaliselt väärtustatud, osaliselt väärtustamata.

Ülesandeid

29. Arvutada interaktiivses keskkonnas i^2.

Õigus on neil, kes nüüd oletavad, et 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)`. Segiminekut tüübipaariga pole karta, kuna Haskellis pole kunagi vaja tüübipaare kirjutada.

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

Ülesandeid

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

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

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

33. Millist assotsiatiivsust, kas vasak- või parem-, oleks võtmesõnal `::` infiksoperaatorina mõttekas omada?

Olulise täpsustusena tuleb märkida, et kõigi infiksoperaatorite väärtuseks loetakse funktsioonid oma *curried*-kujul. Seega näiteks `&&` väärtuseks olev funktsioon mitte ei võta argumendiks 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.

Sellele vastavalt näitab infiksoperaatorite tüüpe ka interaktiivne keskkond. (Kuna paljas infiksoperaator ei moodusta iseseisvat avaldist, tuleb tema tüüpi küsida käsuga `:i`, mitte `:t`.) Näiteks infiksoperaatori `&&` tüübiks annab interaktiivne keskkond `Bool -> Bool -> Bool` ja `+` tüübiks `(Num a) => a -> a -> a`.

Siin `->` on funktsioonitüübikonstruktor, tema väärtuseks on tüübifunktsioon, mis suvalisel kahel tüübil A, B annab tulemuseks tüübi $A \to B$, millesse kuuluvad parajasti funktsioonid argumenditüübiga A väärtustüübiga B. Nagu näha, on ka `->` infiksoperaator ja seejuures paremassotsiatiivne, muidu oleks tulnud kirjutada `Bool -> (Bool -> Bool)` ja `a -> (a -> a)`.

Infiksoperaatori `:` väärtuseks olev funktsioon on tüüpi $A \to \text{List}\ A \to \text{List}\ A$ mistahes tüübi A jaoks. Seega on operaatori `:` tüüp `a -> [a] -> [a]`.

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. *Curried*-kujude variant on eelistatud argumentide võtmisele komplektina sellepärast, et viimane tähendaks Haskellis argumentide paigutamist järjenditesse, kuid andmestruktuuride (isegi paaride) moodustamiseks ja neist väärtuste kättesaamiseks kulub mõnevõrra lisaressurssi.

Ülesandeid

34. Küsida interaktiivses keskkonnas infiksoperaatorite `*`, `/`, `^`, `^^`, `**`, `||`, `==`, `<` ning avaldise `[]` tüübid ja saada neist aru.

Prefiksoperaatorid

Vastanduvalt infiksoperaatoritele võiks funktsiooniväärtusega muutujat või konstruktorit, kui ta kirjutatakse oma argumendi ette, nimetada **prefiksoperaatoriks**. Iseseisvalt käivitatava programmi tegemisel juba kasutasime üht prefiksoperaatorit `print`, mille väärtuseks on funktsioon,

mis argumendil x annab väärtuseks protseduuri, mis kirjutab standardväljundisse stringikujul x. Prefiksne rakendamine on isegi levinum kui infiksne.

Haskellis eraldatakse funktsiooniväärtusega avaldis oma argumendist üldiselt tühikuga, nt kujul `log 5`. Tühiku asemel on lubatud ka muu mittetühi tühisümbolite jada, mis ei tekita rajastusvigu (tühisümboliteks on peale tühiku näiteks tabulaator ja reavahetus). Kui kas operaator või argument on sulgudes, võib neid eraldav tühisümbolite jada isegi tühi olla, nt `log(5)`, `(log)5`, `(log)(5)` on kõik samad mis `log 5`.

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

Moodulis Data.Char on näiteks muutuja `ord` — teisendus sümbolist koodiks, `chr` — tema pöördfunktsioon ehk teisendus koodist sümboliks. Samast moodulist tuleva muutuja `isUpper` väärtuseks on predikaat (st funktsioon, mille väärtustüüp on tõeväärtusetüüp, ehk siis tõeväärtusi välja andev funktsioon), mis kontrollib, kas argument on suurtäht, muutuja `toUpper` väärtuseks on aga funktsioon, mis võtab argumendiks sümboli ja kui see on väiketäht, siis annab väärtuseks vastava suurtähe, muidu aga argumendi enda. Samast moodulist saab palju teisigi analoogseid predikaate ja teisendusfunktsioone.

Prefiksse rakendamise kasutamisel tuleb arvestada, et see seob identifikaatoreid tugevamini igast infiksoperaatorist ning lugema hakatakse vasakult. Teisiti öeldes, kui võtta prefiksset rakendamist infiksoperaatorina, mille argumendid on prefiksoperaator ja tema argument, siis selle infiksoperaatori prioriteet on 10 ja ta on vasakassotsiatiivne.

Ülesandeid

35. Kontrollida ujukomaarvuliste arvutuste täpsust, arvutades arvu π mõne arkusfunktsiooni kaudu ja võrreldes muutuja `pi` väärtusega.
36. Kompleksmuutuja funktsioonide teoorias $i^i = e^{-\frac{\pi}{2}}$. Arvutada see arv interaktiivse interpretaatoriga kahel viisil: kompleksarve kasutades ja kompleksarve kasutamata.
37. Kontrollida ühe avaldise väärtustamisega, kas sümbol koodiga 7 on suurtäht.
38. Katsetades interaktiivses keskkonnas paljude erinevate argumentidega, uurida mooduli Data.Char mõne ülal mitte kirjeldatud muutuja väärtust.
39. Küsida interaktiivses keskkonnas muutujate `log`, `abs`, `not`, `ord`, `isUpper`, `toUpper` tüübid ja saada neist aru.
40. Mis on avaldise `log 0` väärtus?

Paaridega seonduvad moodulis Prelude defineeritud muutujad `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. Kummagi muutuja rakendamisel seisneb arvutus vaid paari järgi komponendi leidmises, komponendi sisusse ei tungita ja ei puututa ka paari seda komponenti, mida välja andma ei pea. Seega on `fst` ja `snd` rakendamine konstantse keerukusega.

Näite paare konstrueerivast funktsioonist saame mooduli Prelude muutujast `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

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

42. Küsida interaktiivses keskkonnas muutujate `fst`, `snd`, `properFraction` tüübid ja saada neist aru.

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

Listide valdkonnas peaks tundma muutujaid `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, avaldise `tail (1 : 0 : [])` väärtus aga 1-elemendiline list $0:[]$. Muutuja `null` väärtuseks on predikaat, mis kontrollib, kas argumentlist on tühi. Kõigi nende muutujate rakendamine toimub konstantse keerukusega argumentlisti pikkuse suhtes.

Moodulis Prelude on defineeritud väga palju listifunktsioone. Kõige lihtsamad neist lisaks seniõpituile on muutujate `length`, `sum` ja `product` väärtuseks. Need funktsioonid võtavad argumendiks listi ja annavad tulemuseks vastavalt oma tema pikkuse ehk elementide arvu, elementide summa ja elementide korrutise. Esimene neist funktsioonidest on korrektselt rakendatav igat tüüpi elementidega listidele, teised kaks aga ainult arvulist tüüpi elementidega listidele. Näiteks avaldise `length (1 : 0 : [])` väärtus on 2, avaldise `product (1 : 2 : 3 : [])` väärtus aga 6. Veel on kasulik tunda muutujat `reverse`, mille väärtuseks on funktsioon, mis võtab argumendiks listi ja annab tulemuseks ümberpööratud listi, kui selline leidub (näiteks lõpmatu listi korral ei leidu). See funktsioon rakendub korrektselt mistahes tüüpi elementidega listidele. Kõigi selles lõigus vaadeldud funktsioonide arvutamine toimub argumentlisti pikkuse suhtes lineaarses ajas.

Ülesandeid

44. Testida viimastes lõikudes käsitletud muutujaid interaktiivses interpretaatoris, koostades igaühe jaoks neist avaldise, milles ta on rakendatud mingile argumendile, mille väärtuseks on vähemalt 3-elemendiline list, nii et avaldise väärtustamine lõpeb normaalselt.

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

46. Milline on muutujate `length`, `sum`, `product`, `reverse` väärtuseks olevate funktsioonide väärtus (a) tühjal listil, (b) lõpmatul listil, (c) osalisel listil?

Kui `head` ja `tail` väärtuseks olevad funktsioonid jagavad listi osadeks tema esimese elemendi järelt, siis muutujate `last` ja `init` väärtuseks on analoogsed funktsioonid, mis jagavad mittetühja argumentlisti l osadeks tema viimase elemendi eest, andes välja vastavalt l viimase elemendi ja listi l elementidest kuni eelviimaseni. Kuid erinevalt muutujatest `head` ja `tail` toimub nende muutujate rakendamine lineaarse keerukusega argumentlisti pikkuse suhtes. Põhjus seisneb selles, et kuna listi elemendid üldiselt ei paikne mälus kompaktselt, vaid lististruktuur ehitatakse mälus üles järjestikuste viitade ahelana, siis viimase elemendi kättesaamiseks on vaja list algusest lõpuni läbi lugeda.

Hulk listifunktsioone, mida moodulist Prelude ei saa, on kättesaadavad standardteegi mooduli Data.List kaudu. Seal on näiteks muutuja `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 muutuja `inits` väärtuseks: tema võtab samuti argumendiks listi ja annab tulemuseks listi, mille elementideks on tema kõik algusjupid alates lühemast: tühi list, 1-elemendiline list esimese elemendiga, list 2 esimese elemendiga jne, kuni kogu listini. Mõlemad funktsioonid töötavad ka tühjal listil.

Kui `tails` rakendamine toimub lineaarse keerukusega argumentlisti pikkuse suhtes, siis `inits` rakendamine nõuab koguni ruutkeerukust. 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. Arvutada interaktiivses keskkonnas list, mille elementideks on listi $1:3:6:10:15:21:28:[]$ kõik mittetühjad algusjupid pikkuse järgi kasvavas järjestuses.

48. Arvutada interaktiivses keskkonnas list, mille elementideks on listi $1:3:6:10:15:21:28:[]$ kõik mittetühjad lõpujupid pikkuse järgi kasvavas järjestuses. Püüda teha nii efektiivselt, kui senised teadmised võimaldavad.

Ka prefiksoperaatori väärtus võib olla *curried*-kujuline, st süüa pärast üht argumenti veel argumente, kuni tulemuseks olev väärtus pole enam funktsioon. Valdav enamik mitmeparameetrilisi

funktsioone standardteegis ongi realiseeritud *curried*-kujulistena, olgu nad siis antud infiks- või prefiksoperaatorite väärtuseks.

Üks lihtsaimaid võimalikke *curried*-kujulisi funktsioone on muutuja `const` väärtuseks. See annab oma argumendil x väärtuseks funktsiooni, mis suvalisel oma argumendil annab väärtuseks x. Teisiti öeldes, `const` väärtus on `fst` väärtuse *curried*-kuju.

Muutuja `const` rakendamiseks argumendile `5` kirjutame, nagu tavalistegi funktsioonide puhul, `const 5`. Selle avaldise väärtuseks on funktsioon, seega saame veel korra rakendada — näiteks argumendile `0.2`, saades avaldise `(const 5) 0.2`. Vastavalt `const` ülalkirjeldatud definitsioonile on viimase avaldise väärtuseks 5.

Taolises kirjutises nagu `(const 5) 0.2` võib sulud ära jätta, kuna funktsiooni rakendamine järjestkirjutamisega on vasakassotsiatiivne, st `a b c = (a b) c`. See on veel üks omadus, mis teeb *curried*-kujul funktsioonide kasutamise Haskellis mugavaks.

Äsjakäsitletud 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`, mis on normaalne väärtus. Argumendi `undefined` antav potentsiaalne veateade jääb olemata, järelikult seda argumenti ei püütagi väärtustada.

Arvulistest funktsioonidest on *curried*-kujulised muutujate `div` ja `mod` väärtused. Need funktsioonid võtavad ükshaaval argumentidena kaks arvu ja annavad väärtuseks vastavalt täisarvulise jagatise ja jäägi esimese arvu jagamisel teisega. Näiteks avaldiste `div 10 7` ja `mod 10 7` väärtuseks 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, võib kasutada muutujat `divMod`, mille väärtuseks on funktsioon, mis annab välja oma argumentide jagatise ja jäägi ühes paaris koos; näiteks `divMod 10 7` väärtuseks on paar $(1\,,3)$.

Veel võib näiteks tuua moodulist Data.Complex saadava muutuja `mkPolar`, mille väärtuseks on *curried*-kujul funktsioon, mis võtab argumendiks ujukomaarvud r ja x ja annab väärtuseks kompleksarvu polaarkoordinaatidega (r, x).

Listide poolelt võiks näiteks tuua `take` ja `drop`. Kummagi muutuja väärtuseks on funktsioon, mis võtab argumendiks lühikese täisarvu n ja annab välja funktsiooni, mis võtab argumendiks mingi 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 : [])` väärtuseks on 2-elemendiline list $5:6:[]$ ja `drop 2 (5 : 6 : 7 : [])` väärtuseks on 1-elemendiline list $7:[]$. Analoogselt jäägiga jagamisega on ka siin olemas võimalus arvutada mõlemad osad korraga, selline funktsioon on muutuja `splitAt` väärtuseks. Seega `splitAt 2 (5 : 6 : 7 : [])` väärtuseks on paar $(5:6:[], 7:[])$.

Muutujate `take`, `drop` ja `splitAt` rakendamine toimub lineaarse keerukusega nende

täisarvargumendi suhtes. Kui listi elementide arv on suurem, ei puutu ülejäänud elemendid asjasse.

Veel läheb tihti vaja muutuja `replicate` väärtuseks olevat *curried*-kujul funktsiooni, mis võtab argumendiks lühikese täisarvu a suvalise objekti x ning annab välja listi pikkusega a, mille iga element on x. Selle muutuja rakendamine toimub lineaarse keerukusega täisarvargumendi suhtes.

Ülesandeid

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

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

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

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

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

Kõik senivaadeldud prefiksoperaatorid on muutujad. Kasutatavaim konstruktor, mis noolib argumendi ja mida kirjutatakse oma argumendi ette, on `Just`. Tema väärtus on funktsioon, mis võtab argumendiks suvalise väärtuse x mingist tüübist A ja annab tulemuseks väärtuse $\mathrm{Just}\, x$ tüübist $\mathrm{Maybe}\, A$. Tüübiperega Maybe seondub veel konstruktor `Nothing`, mis pole funktsioon.

Ülal mainisime muutujat `head`, mille väärtuseks on listi järgi tema pea leidmise funktsioon. See funktsioon töötab listitüübist vastavasse elemenditüüpi. Kui argumendiks juhtub tulema tühi list, siis pead ei ole, pole võimalik midagi vastuseks anda ja `head []` väärtustamine lõpetab täitmisaegse veaga. Standardteegi moodulist Data.Maybe leiab aga muutuja `listToMaybe`, mille väärtuseks on funktsioon, mis töötab listitüübist Maybe-tüüpi: kui argumentlist on mittetühi ja tema pea on x, siis annab see funktsioon väärtuseks $\mathrm{Just}\, x$. Kui argument on tühi, siis tuleb väärtuseks Nothing. See funktsioon on sisult sarnane, kuid ei tekita ühelgi juhul täitmisaegset viga.

Kui on kindel, et avaldise väärtus pole Nothing, võib konstruktorist `Just` lahtisaamiseks rakendada muutujat `fromJust` moodulist Data.Maybe. Selle muutuja väärtuseks on funktsioon, mis argumendil $\mathrm{Just}\, x$ annab tulemuseks väärtuse x ja argumendil Nothing lõpetab veaga. Samast moodulist saab veel palju funktsioone, mis seonduvad Maybe-tüüpidega.

Eelnevalt mainisime funktsiooni rakendamist märkiva järjestkirjutamise tinglikku käsitlust infiksoperaatorina. Märgime, et prefiksse rakendamise jaoks on olemas ka päris infiksoperaator `$`. Näiteks avaldise `log 5` kohal võib samaväärselt kirjutada `log $ 5`.

Operaatori `$` prioriteet ja assotsiatiivsus on tavalise järjestkirjutamisega võrreldes vastupidised: ta on paremassotsiatiivne prioriteediga 0. Seega on `$` kasulik olukorras, kus mingile objektile rakendatakse järjest mitut funktsiooni. 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.

Suundume nüüd tüübimaailma ja otsime näiteid prefiksoperaatoritest tüüpidel. Loomulik mõte on, et tüübifunktsioonid, mille liik on $* \to *$, nagu näiteks List, Maybe ja IO, võiksid olla realiseeritud prefiksoperaatoritena. See oletus peab ka paika: tüübifunktsiooni List kodeerib Haskellis kirjutis `[]`, tüübifunktsioone Maybe ja IO aga vastavalt sõnad `Maybe` ja `IO`. Näiteks 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 operaator `[]` on erandliku kujuga — ta ei vasta leksikanõuetele —, ei käsitle interaktiivsed keskkonnad kasutajale vastuseid koostades teda üldse prefiksoperaatorina, vaid täiesti erilaadsena. Nimelt asetavad interaktiivsed keskkonnad operaatori `[]` argumendi kantsulgude sisse, mitte välja järele.

Ülesandeid

54. Küsida interaktiivses keskkonnas muutujate `head`, `tail`, `null`, `length`, `sum`, `reverse`, `tails`, `inits`, `take`, `drop`, `splitAt`, `replicate` tüübid ja saada neist aru.

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

56. Kirjutada korrektne tüübiga annoteeritud avaldis, mille tüüp on listitüüp, mille elemenditüüp on mingi Maybe-tüüp.

Prefiksoperaatorid tulevad mängu ka kompleksarvutüübi juures. Nimelt sõltub kompleksarvutüüp sellest, millist ujukomaarvutüüpi on väljad. Moodulis Data.Complex defineeritud tüübikonstruktori `Complex` väärtuseks on funktsioon, mis võtab argumendiks väljatüübi A (see peab

olema klassist Floating) ja annab tulemuseks kompleksarvutüübi, mille väljatüüp on A. Näiteks avaldise `1 :+ 0` tüübiks võib kirjutada `Complex Float` või `Complex Double` või `(Floating a) => Complex a`.

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

On defineeritud tüübimuutuja `Rational`, mis on ekvivalentne avaldisega `Ratio Integer` ja mis on nähtav ka mooduli Prelude kaudu. Tegemist on globaalse tüübimuutujaga ja selliste nimed peavad algama suurtähega. Tavaliselt nimetatakse sellist tüübimuutujat **tüübisünonüümiks** (ingl *type synonym*).

Ülesandeid

57. Uurida, kuidas interaktiivne keskkond näitab ratsionaalarve. Sooritada mõned tehted ratsionaalarvudega, kus osaleksid ka murdarvud.

Nime üldkuju ja infikskuju

Infiksoperaatorite tüüpide seletamisel märkisime möödaminnes, 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 paradigma rikkumist pole. Asi on selles, et kõigil andmemuutujatel ja -konstruktoritel on põhimõtteliselt olemas kaks nimekuju: üks infiksseks ja teine muuks kasutamiseks. Ütleme nende kujude kohta vastavalt **infikskuju** ja **üldkuju**.

Sümbolitest (2) koosnevad nimed on kõik infikskujul ja et neid kasutada mitteinfiksselt, tuleb nad asetada sulgudesse. Näiteks `*` üldkuju on `(*)`. Tähtedest, numbritest ja alakriipsust koosnevad nimed on üldkujul ja et neid kasutada infiksselt, tuleb nad panna tagurpidiülakomade vahele. Näiteks `mod` infikskuju on `` `mod` ``.

Mistahes muutuja või konstruktor moodustab omaette avaldise küll, aga ta tuleb selleks esitada üldkujul, sest omaette avaldisena esitades pole kasutus infiksne. Näiteks `*` ei ole avaldis, kuid `(*)` on.

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.

Täpsemalt, muutuja või konstruktor on reaalselt infiksselt kasutatav parajasti siis, kui tema väär-

tuseks on *curried*-kujul funktsioon. Sellisel juhul vastab prefiksse rakendamise esimene argument infiksse rakendamise vasakule argumendile ja prefiksse rakendamise teine argument infiksse rakendamise paremale argumendile. Teistsuguse muutuja või konstruktori infiksne kasutus annab tüübivea.

Vaatleme näiteks infiksoperaatorit `++`, mille väärtuseks on binaarne konkatenatsioon (ingl *concatenation*), st funktsioon, mis võtab argumentideks kaks listi l ja k, mille elemendid on sama tüüpi, ja annab välja listi, mille alguses on järjest kõik l elemendid ja kui l on lõplik, siis sellele järgnevad järjest kõik k elemendid. Näiteks avaldise `replicate 3 0.5 ++ replicate 4 0.8` väärtus on list $0.5 : 0.5 : 0.5 : 0.8 : 0.8 : 0.8 : 0.8 : []$. Kuid sama avaldis on esitatav ka kujul `(++) (3 ‘replicate‘ 0.5) (4 ‘replicate‘ 0.8)`, kus varasem infiksoperaator `++` on kasutatud prefiksselt ja varasem prefiksoperaator `replicate` infiksselt.

Konkatenatsiooni arvutuse käigus ehitatakse parempoolse listi ette vasakpoolse koopia, parempoolset listi ei uurita üldse. See tähendab, et konkatenatsiooni teostamine sõltub lineaarselt vasakpoolse listi pikkusest ja ei sõltu parempoolse listi pikkusest. Sellel põhjusel on konkatenatsioon tehtud paremassotsiatiivseks, sest nii on arvutus efektiivsem. Arvutades avaldist `(a ++ b) ++ c`, kopeeritakse kõigepealt list `a` poolt märgitav list `b` poolt märgitava listi ette ja seejärel mõlemad koos `c` poolt märgitava listi ette. Esimest listi kopeeritakse kaks korda. Sulgude parempoolse paigutuse korral kopeeritakse iga elementi ülimalt üks kord. Efektiivsuse vahe on seda suurem, mida rohkem liste järjest tuleb konkateneerida.

Kõik infikssed rakendamised kirjutab Haskelli süntaksianalüsaator samaväärselt prefikssete rakendamiste kaudu ümber. Näiteks avaldisest `a * b` saab `(*) a b` ja avaldisest `a ‘mod‘ b` saab `mod a b`.

Prefiksoperaatorist tagurpidiülakomade abil tuletatud infiksoperaatorid on, nagu ülejäänudki, vaikimisi prioriteediga 9 vasakassotsiatiivsed, kuid muutujate `div` ja `mod` infikskujud on korrutamise-jagamisega sama prioriteediga 7 vasakassotsiatiivsed. Haskell-programmis on võimalik oma defineeritavate nimede infikskujude vaikeatribuute muuta, kuid selleni jõuame hiljem.

Pangem tähele, et nimesid imporditakse ja eksporditakse ainult oma algkujul. Moodul Data.Complex näiteks ekspordib nimekuju `:+`, kuid mitte nimekuju `(:+)`, ning ekspordib nimekuju `mkPolar`, kuid mitte kuju `‘mkPolar‘`. Kuju teisendamine on puhtlokaalne ettevõtmine. Nime adresseerimine tema kuju ei mõjuta, nt `Data.Ratio.%` on infikskuju ja talle vastav üldkuju on `(Data.Ratio.%)`.

Ülesandeid

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

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

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

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

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

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

Mis puutub tüübimuutujatesse ja -konstruktoritesse, siis infiksoperaatoritel on siingi üldkuju olemas. Näiteks funktsioonitüüpi Int → Char saab Haskellis kirjutada nii kujul `Int -> Char` kui ka kujul `(->) Int Char`. Infikskuju moodustamine tähtedest, numbritest ja alakriipsust koosneva nimega tüübioperaatoritest on standard-Haskellis keelatud. GHC laiendustes on aga ka see võimalik.

Sektsioonid

Andmemaailma infiksoperaatoritest saab spetsiaalse süntaktilise konstruktsiooniga moodustada nn **sektsioone** (ingl *section*), mis kujutavad endast avaldist, mille väärtus on funktsioon ja mida rakendatakse prefiksselt. Sektsioonid jagunevad vasak- ja paremsektsioonideks.

Vasaksektsioon (ingl *left section*) näeb välja kujul `(𝔞 ⊕ )`, kus `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 `𝔞` väärtus. Analoogselt näeb **paremsektsioon** (ingl *right section*) välja kujul `( ⊕ 𝔟)`, kus $\oplus$ on infiksoperaator ja `𝔟` avaldis. Sellise avaldise väärtus on funktsioon, mis igale argumendile a seab vastavusse $f\,a\,b$, kus f on $\oplus$ väärtus ja b on `𝔟` väärtus.

Seega sektsiooni rakendamine argumendile on mõlemal juhul 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. Mõlemat laadi sektsioone rakendatakse prefiksselt.

Näiteks `(* 2)` on funktsioon, mis korrutab oma argumendi 2-ga. Sama väärtusega on ka `(2 *)`, sest korrutamine on kommutatiivne. Avaldised `(/ 2)` ja `(2 /)` on aga erineva väärtusega: esimene on funktsioon, mis jagab oma argumendi 2-ga, teine aga funktsioon, mis jagab arvu 2 oma argumendiga. Nii on näiteks `(* 5) 7` väärtuseks 35, `(3 ^) 2` väärtuseks 9 ja `(^ 3) 2` väärtuseks 8.

Kui vasaksektsioonide moodustamisel infiksoperaatoreist piiranguid pole, siis paremsektsiooni moodustamine miinusest pole lubatud, sest kirjutis kujul `(- 𝔞)` läheks segi negatiivse arvuliteraaliga `-𝔞`. Olukorras, kus oleks vaja miinuse paremsektsiooni, aitab hädast välja muutuja

`subtract`, mille väärtuseks on *curried*-kujul 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

64. Kirjutada ülesannetes 58 ja 59 toodud avaldistes infikssed rakendamised vasaksektsioonidega ümber.
65. Kirjutada ülesannetes 58 ja 59 toodud avaldistes infikssed rakendamised paremsektsioonidega ümber.
66. Selgitada võimalikult lihtsate sõnadega, mis on avaldise `(: [])` väärtus.
67. Teha kindlaks, kas avaldiste

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

    väärtuseks olevad funktsioonid on agarad või laisad.

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

Vaatleme näiteks muutujat `map`, mille väärtuseks on *curried*-kujul 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. Kui meil on mingi avaldis `l` tüüpi `[Integer]` ja tahame arvutada tema väärtuse kõigi elementide ruudud, siis kirjutis `map (^ 2) l` on selleks lihtsaim tee.

Tähelepanelik lugeja märkas, et `map` väärtuseks on kõrgemat järku funktsioon: ta võtab argumendiks funktsioone.

Ülesandeid

68. Lahendada ülesanne 26, kasutades sektsioone ja muutujat `map`.
69. Küsida interaktiivses keskkonnas muutuja `map` tüüp ja saada sellest aru.

Listide erisüntaksid

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

Listitüübid, elementide loendid, stringikonstandid

Oleme näinud, et interaktiivsed keskkonnad esitavad kasutajale listitüüpe mitte reeglipärasel kujul, vaid kujul, kus listitüübikonstruktori `[]` argument on konstruktori sees kantsulgude vahel.

See esitus kuulub ka Haskelli süntaksisse, Haskell lubab tüübioperaatorit `[]` lisaks prefikssele kasutusele rakendada ebareeglipäraselt ülalkirjeldatud viisil. Tähenduse vahet võrreldes prefiksse rakendamisega pole.

Niisiis on näiteks `[Integer]` ja `[Bool]` tüübiavaldised, mis väljendavad tüüpe, kuhu kuuluvad vastavalt täisarvuliste elementidega listid ja tõeväärtustüüpi elementidega listid. Kuna stringid on sümbolite listid, tähendab stringitüüpi tüübiavaldis `[Char]`; kuid on olemas ka ekvivalentne tüübimuutuja `String`.

Samuti oleme näinud, kuidas interaktiivsed keskkonnad esitavad kasutajale mittetühje liste: komaga eraldatud elementide loendina kantsulgudes. Ka see kuulub tegelikult Haskelli süntaksisse.

Näiteks **`[`**`pi`**`]`** on avaldisega `pi : []` samaväärne avaldis, **`[`**`log 1, log 3, log 5`**`]`** aga avaldisega `log 1 : log 3 : log 5 : []` samaväärne avaldis.

Selle süntaksiga 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.

Kuna stringid on listid, on stringitüüpi objekte võimalik kodeerida kõigi sobivate listisüntaksite abil. Stringikonstandid on siiski kodeeritavad üldlevinud kirjaviisiga — sümbolite joruga jutumärkide vahel. See on sisseehitatud erisüntaks sümbolite listi jaoks. Seejuures on kasutatavad kõik sümbolite langjoonkoodid (ainult et stringis pole neid muidugi vaja apostroofide vahele panna). Apostroofi kodeerimine kujul `\'` pole kohustuslik, küll aga jutumärgi kodeerimine kujul `\"`.

Stringikonstant ei saa sisaldada muutujaid. Stringikonstandi kasutamine seega eeldab, et terve string koos oma sümbolitega on kodeerimise ajal teada.

Ülesandeid

70. Kirjutada interaktiivse interpretaatori käsurealt avaldisi, mille väärtuseks on string “Tere!”. Kasutada kolme erinevat süntaktilist konstruktsiooni.

Java `toString`-meetodiga analoogse Haskelli muutuja `show` väärtus on funktsioon, mis annab välja oma argumendi stringikujul. See muutuja on defineeritud arvude, sümbolite, tõeväärtuste jpt, üldiselt klassi Show kuuluvate tüüpide esindajate jaoks, kuid mitte kõigil Haskellis ette tulevatel objektidel.

Näiteks funktsioonide stringiks teisendamine on standardteegis eeldefineerimata. Samuti võib juhtuda, et `show` väärtuseks on defineeritud funktsioon, mille tulemus ei kajasta täpselt argumendi originaalset struktuuri. Nii on lugu näiteks listidega, mis teisendatakse enamasti argumentide komadega eraldatud nimekirjaks kantsulgude vahel, sarnaselt erisüntaksile. Stringid aga pannakse jutumärkidesse ja asendatakse temas leiduvad erisümbolid nende langjoonkoodidega. Kodeerimist kasutatakse ka üksikute sümbolite korral. Kõigil loetletud juhtudel kehtib põhimõte, et stringiksteisendamise tulemus on ise korrektne Haskelli kirjaviis antud väärtuse esitamiseks.

Muutuja `show` rakendamise toimet saab interaktiivses keskkonnas testida isegi ilma teda ilmutatult rakendamata, sest interaktiivsed keskkonnad rakendavad just teda vaikimisi kõigile avaldistele, mida kasutaja neil väärtustada laseb, et nende väärtusi väljastamiseks stringikujule viia. Hugsis on võimalik see käitumine vajadusel ära muuta, võttes maha optsiooni `u` käsuga `:s -u` — siis teisendab Hugs kõik avaldised stringiks sisseehitatud viisil, sõltumata `show` väärtusest.

Vaikiv `show` lisamine interaktiivses keskkonnas põhjustab selle, et täiesti korrektse avaldise andmisel käsurealt tekib tüübiviga, kui see avaldis on tüüpi, mille jaoks on muutuja `show` defineerimata.

Ülesandeid

71. Anda 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?

72. Loetleda järjest sümbolid avaldise `show 3`, avaldise `show (show 3)` ja avaldise `show (show (show 3))` väärtuseks olevas stringis.

73. Anda interaktiivse interpretaatori käsurealt avaldisi, mille väärtus on funktsioon, ja saada aru järgnevast veateatest.

74. Saavutada Hugsis olukord, kus ta funktsioonitüüpi avaldise väärtustamisel ei anna veateadet.

Muutuja `print` rakendamisel mingile argumendile rakendatakse kõigepealt muutujat `show` ja seejärel kirjutatakse saadud string standardväljundisse. Seega `print` on kasutatav ainult sellistel argumentidel, millel ka `show`. Kui objekt, mida soovitakse standardväljundisse saata, juba on string, siis võib olla sobivam `print` asemel kasutada muutujaid `putStr` ja `putStrLn`. Nende muutujate väärtuseks on funktsioonid, mis võtavad argumendiks stringi ja kirjutavad ta standardväljundisse, `putStrLn` seejuures lisab ka reavahetuse.

Stringikonstantide võibolla et levinuim kasutus on veateadete kirjutamisel. Veateadete tekitamiseks on olemas operaator `error`, mille väärtus on funktsioon, mis võtab argumendiks suvalise stringi s ja annab tulemuseks $\perp$; selle 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 suvalist tüüpi, seetõttu on niimoodi võimalik täitmisaegset viga tekitada ühtmoodi sõltumata sellest, millist tüüpi tulemust kontekst nõuab.

Ülesandeid

75. Küsida interaktiivses keskkonnas muutuja `error` tüüp ja saada sellest aru.
76. Anda interaktiivse keskkonna käsurealt korrektne avaldis tüüpi `Int`, mille väärtustamisel lõpetatakse töö teie ette antud veateatega.

Aritmeetilised progressioonid

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

Üldine tõkestatud progressioon esitatakse kahe esimese elemendi ja tõkke kaudu kujul `[a, b .. c]`. Kui `a`, `b`, `c` väärtuseks on vastavalt arvud a, b, c, siis `[a, b .. c]` väärtus on list, mille elemendid võetakse järjest aritmeetilisest jadast esimese elemendiga a vahega $b - a$ parajasti senikaua, kui jada liikmed pole arvteljel läinud arvust c teisele poole arvu $c - (b - a)$. Näiteks avaldis `[1, 3, 5, 7, 9]` on samaväärne avaldisega `[1, 3 .. 9]`, samuti avaldisega `[1, 3 .. 10]`.

Kui progressioon on sammuga 1, võib aritmeetilise jada süntaksis koma ja talle järgneva elemendi ära jätta. Näiteks avaldise `[1 .. 10]` väärtuseks on list järjekorras arvudest 1 kuni 10.

Ülesandeid

77. Koostada aritmeetilise jada süntaksiga avaldisi ja anda neid interaktiivse interpretaatori käsurealt. Proovida nii positiivse kui negatiivse vahega progressioone ning kummalgi juhul nii esimesest elemendist suurema kui väiksema väärtusega tõkkega.
78. Arvutada kõigi 3-kohaliste 1-ga lõppevate arvude korrutis.
79. Arvutada suurim 2006-ga jaguv arv esimese 100000000 naturaalarvu seas, lastes selleks väärtustada aritmeetikatehteid mitte sisaldava avaldise.
80. Kirjutada interaktiivses keskkonnas avaldis, mille väärtuseks on list, milles on kasvavas järjestuses parajasti kõik positiivsed täisarvud kuni 1000-ni peale paaritute kolmekohaliste.

Haskellis võib avaldise väärtus olla ka lõpmatu andmestruktuur, muuhulgas lõpmatu list. Selliseid avaldisi saab praktikas edukalt kasutada, sest liste konstrueeritakse laisalt, parajasti niipalju

kui hetkel vaja. Asjaolu, et avaldise väärtuseks on lõpmatu list, tähendab seda, et avaldise asetamisel sobivasse konteksti on põhimõtteliselt võimalik lõpliku ressursiga arvutada selle listi kuitahes kauget fragmenti. Sobiv kontekst peab täpsustama, millist osa listist tahetakse. Lõpmatu listi enda täielik väärtustamine on muidugi ükskõik kui suure lõpliku ressursiga võimatu.

Lõpmatud listid on mugavad, sest võimaldavad tihti vältida ülesande seisukohalt kunstliku piiri sissetoomist.

Kui proovisite aritmeetilise progressiooni süntaksi harjutamisel kirjutada progressiooni, milles vahe on 0 ja tõke pole esimesest elemendist väiksem, näiteks avaldisega `[0, 0 .. 1]`, siis olete lõpmatu listiga juba vastamisi sattunud. Lõpmatu listi täieliku väärtustamise protsess ei peatu enne, kui miski väline teda katkestab. Interaktiivses keskkonnas saab protsessi katkestada, vajutades <*Ctrl*>+`c`. Parem variant lõpmatu listi testimiseks on tema erinevaid lõplikke osi küsida.

Lõplike algusjuppide küsimiseks sobib juba varem vaadeldud muutuja `take`. Teine võimalus on pärida üksikuid elemente, milleks 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 avaldise `(1 : 2 : 3 : []) !! 1` väärtus on 2. Avaldise `[0, 0 .. 1] !! a` väärtus on alati 0, kui vaid `a` väärtuseks on naturaalarv.

Tuleb arvestada, et listi elemendi leidmine operaatoriga `!!` toimub järjekorranumbri suhtes lineaarses ajas. Lineaarne sõltuvus järjekorranumbrist tekib listi ahelalaadse ülesehituse tõttu, kus iga elemendini on võimalik jõuda vaid listi läbikäimisega algusest temani.

Kuna lõpmatud listid on praktilised, siis pole midagi imestada, et aritmeetilise progressiooni süntaksist leiduvad Haskellis ka tõkestamata variandid. Need saame senivaadeldutest, kui jätame tõket märkiva avaldise ära. Sellisel juhul on avaldise väärtuseks lõpmatu 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 .. ]` väärtuseks on list kõigi paaritute naturaalarvudega kasvavas järjestuses, `[1 .. ]` väärtuseks aga list kõigi positiivsete naturaalarvudega kasvavas järjestuses.

Ülesandeid

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

82. Väärtustada interaktiivse interpretaatori käsurealt avaldis, mille väärtuseks on lõpmatu list. Katkestada arvutus ja seejärel täiendada avaldist nii, et saadud avaldise väärtustamine lõpeb kiiresti.

83. Kirjutada interaktiivses keskkonnas avaldis, mille väärtuseks on lõpmatu list, mille esimesed elemendid on 1 ja 2 ning kõik ülejäänud elemendid on 0-d. Testida seda listi tema

üksikuid elemente küsides.

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

85. Kirjutada ühele reale mahtuv avaldis, mille väärtuseks on list, milles on kasvavas järjestuses 12 esimest 899979996999599949993999299919990999-ga jaguvat positiivset täisarvu.

86. Katseliselt veenduda, et `!!` abil listi elemendi leidmine on järjekorranumbri suhtes lineaarse keerukusega.

Aritmeetilise progressiooni süntaksi kasutamiseks ei pea listide elemendid tingimata arvud olema. Nii saab koostada liste, mille elementide tüüp on suvaline klassi Enum esindaja. Tõlgendamaks aritmeetilise progressiooni avaldist mittearvuliste elementide puhul teisendatakse need elemendid lühikesteks täisarvudeks funktsiooniga, mis on muutuja `fromEnum` väärtuseks, koostatakse list aritmeetilise progressiooniga nende baasil ja seejärel teisendatakse listi elemendid tagasi algsesse tüüpi funktsiooniga, mis on muutuja `toEnum` väärtuseks.

Muutujad `fromEnum` ja `toEnum` omavad tähendust parajasti klassi Enum tüüpide jaoks. Nende väärtust võib mõista tüübiteisendusfunktsioonina või ka väärtuste kodeerimis- ja dekodeerimisfunktsioonina, kusjuures koodid on lühikesed täisarvud, nii nagu juba varem käsitletud `ord` ja `chr` moodulist Data.Char vastavalt kodeerivad ja dekodeerivad sümboleid. Enamgi, sümbolitüüp kuulub klassi Enum, kusjuures `fromEnum` ja `toEnum` ongi väärtuselt võrdsed just nendesamade muutujatega `ord` ja `chr`.

See näitab esiteks, et aritmeetilise progressiooni erisüntaksit saab kasutada ka stringide esitamiseks. Teiseks, sümbolite järjestus vastab nende koodide kui arvude standardsele järjestusele. Kolmandaks tuleneb siit, et sümbolite koodide kasutamiseks pole moodulit Data.Char laadida vajagi, kuna `fromEnum` ja `toEnum` saab ka moodulist Prelude. Kuid `toEnum` kasutamisel võib olla vaja annoteerida tulemustüüp, kui kontekstist pole see tüüp üheselt selge. Näiteks tahtes leida sümbolit koodiga 33, tuleb kirjutada `toEnum 33 :: Char`, sest muidu süsteem ei tea, millisesse Enum-klassi tüüpi on vaja teisendada.

Ülesandeid

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

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

89. Teha kindlaks tõeväärtuste täisarvkoodid.

90. Arvutada interaktiivses keskkonnas kiiresti ladina tähestiku tähtede arv.

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

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 elemente ei vasta. Sellisel juhul sellel kohal list lõpetatakse. Seega tõkestamata aritmeetilise progressiooni süntaks võib väärtuseks ka lõpliku listi anda.

Näiteks `[False .. ]` väärtuseks on 2-elemendiline list False : True : [].

Kõige universaalsem moodus lõpmatut listi tekitada on mitte aritmeetilise progressiooni süntaksiga, vaid hoopis muutujaga `repeat`. Selle muutuja väärtuseks olev funktsioon võtab suvalise argumendi x ja annab väärtuseks lõpmatu listi, mille iga element on x. See sobib ka juhul, kui x tüüp ei kuulu klassi Enum. Kui vaja huvitavamat listi, võib kasutada muutujat `cycle`, mille väärtuseks on funktsioon, mis võtab argumendiks listi l ja kui see pole tühi, siis annab välja lõpmatu listi, kus l elemendid korduvad tsükliliselt. Näiteks `cycle [1, 2, 1]` väärtuseks on lõpmatu list elementidega $1, 2, 1, 1, 2, 1, 1, 2, 1, \ldots\ldots$.

Haskellis programmeerides puutume tihti kokku mitte ainult eraldiseisvate lõpmatute listidega, vaid andmestruktuuridega, mille komponentideks on lõpmatud listid. Näiteks avaldise `([1, 3 .. ] , [2, 4 .. ])` väärtus on paar, mille kumbki komponent on lõpmatu list. Testides niisugust andmestruktuuri, on oht osa kogustruktuurist ära kaotada. Näiteks kui laseksime seda paari lihtsalt käsurealt väärtustada, näeksime ainult paarituid arve, arvutus ei jõuaks kunagi paari teise komponendini. Ometi on ka paarisarvud olemas, selles veendumiseks piisab lihtsalt võtta struktuurist teine komponent: `snd ([1, 3 .. ] , [2, 4 .. ])` annab tulemuseks paarisarvudest koosneva listi.

Ülesandeid

92. Kirjutada interaktiivses keskkonnas avaldis, mille väärtuseks on lõpmatu list, mille iga element on kõigi naturaalarvude list. Testida seda lõpmatut maatriksit tema üksikute elementide küsimisega.

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

Oma muutujate kasutamine

Lihtsad konstruktsioonid näidistega

Süntaktilised objektid, mida oleme seni vaadelnud, on konstrueeritud vaid mujal defineeritud muutujate ja konstruktorite kaudu; uute muutujate ja konstruktorite defineerimist me veel vaadelnud pole. (Erandiks olid polümorfsete avaldiste tüübid, milles sisalduvad lokaalsed tüübimuutujad.)

Uute muutujate defineerimiseks on vaja kasutada näidiseid — avaldistele tähenduselt vastanduvaid süntaktilisi objekte. Nemad moodustavad käesoleva jaotise põhiteema. Enam ei saa mööda ka deklaratsioonidest. Käesolevas jaotises piirdume lihtsate näidiseid kasutavate deklaratsioonide ja avaldistega, milles puudub hargnemine.

Konstruktorite defineerimine on hoopis teistsugune asi ja jääb esialgu vaatluse alt välja. Vahe on selles, et muutuja defineerimisel seotakse muutuja juba põhimõtteliselt olemasoleva väärtusega, kuid konstruktori defineerimine tähendab väärtuste maailma laiendamist.

Avaldised ja näidised

Enne kui selgitada näidiste ja deklaratsioonide olemust, vaatame korraks üldplaanis peale ka senituntud struktuursetele süntaktilistele objektidele — avaldistele.

Avaldis esitab skeemi, kuidas üks väärtus — avaldise `väärtus` — sõltub avaldise elementaarosade 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 väärtus sõltub muutujate `x`, `y` ja `+` väärtusest.

Teisiti öeldes, avaldis esindab väärtuste laine liikumist “seest välja”.

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

väärtustest.

Avaldise teisendamist info hankimiseks tema väärtuse kohta nimetatakse avaldise **väärtustamiseks** (ingl *evaluation*).

Näidis (ingl *pattern*) kujutab endast avaldise suhtes tähenduse poolest duaalset süntaktilist objekti. Näidis esitab skeemi, kuidas näidise elementaarosade väärtused sõltuvad ühest suvalisest (sobivat tüüpi) väärtusest — näidise **väärtusest**. Näidis esindab väärtuste laine liikumist "väljast sisse".

Näiteks kui näidise `(a , b)` väärtus on paar $(2, 3)$, siis `a` väärtus on 2 ja `b` väärtus on 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 ühest etteantud väärtusest. Nagu näha, võivad avaldis ja näidis ühtmoodi välja näha. Kontekst näitab, kas mõtteks on konstrueerida väärtus etteantud muutujate väärtuste järgi või leida muutujate väärtused etteantud väärtuse järgi.

Sellises vaates avaldistele ja näidistele käsitletakse muutujate väärtusi muutuvatena, kuid konstruktorite väärtusi konstantsetena. Näiteks sulud ja koma tähendavad üheselt paarikonstruktorit, kuid `+` väärtus võis varieeruda. See näitab, et avaldise ja näidise "elementaarosa", millele sai viidatud, ei tähenda siin igasugust süntaktiliselt atomaarset osa: konstruktorid, olgugi et süntaktiliselt atomaarsed, jäävad mängust välja, nemad saavad oma väärtuse väljaspool.

Oleme tõele lähedal, kui ütleme, et elementaarosadeks on parajasti kõik muutujad. Avaldiste puhul selline arusaam töötab ja töötaks ka näidiste puhul, kui me Haskelli asemel tegeleks loogikaga, mille semantikas puudub arvutusprotsess. Haskell aga võimaldab väärtustamise järjekorda varieerida, märkides suvalisi näidiseid vaadeldavas mõttes elementaarseteks. Niisuguse näidise juures väärtuste laine progresseerumine peatub, sissepoole ei jätku.

Sõltumata näidise konstruktsioonist kehtib nõue, et ükski muutuja ei tohi näidises korduda. Näiteks `(a , a)` on näidisena illegaalne. Põhjus on selles, et niisugune näidis ei võimalda muutujate väärtusi üheselt määrata. Kui tema väärtus oleks $(2, 3)$, mida ei saa välistada, siis `a` väärtus peaks olema nii 2 kui ka 3.

Isegi kui näidis on legaalne, võib juhtuda, et tema kasutamine muutujate või teiste elementaarosade väärtuste määramiseks pole võimalik. Elementaarosade väärtuste määramiseks nõutakse, et näidis ja väärtus klapiksid omavahel oma struktuuri poolest. Seda, et näidis ja väärtus klapivad struktuurilt, nimetatakse näidise **sobitumiseks** (ingl *matching*) väärtusega. Kui näidis sobitub väärtusega, siis väärtus määrab näidise kõigi elementaarosade väärtused üheselt.

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. Sama näidis siiski sobitub väärtusega $(2, \bot)$ ja isegi väärtusega $(\bot, \bot)$.

Vaatame veel näidist `x : xs`. Tema määrab muutujate `x` ja `xs` väärtused mistahes mittetühja listi järgi: kui näidise väärtus on näiteks $1 : 0 : []$, siis `x` väärtuseks on 1 ja `xs` väärtuseks $0 : []$, kui aga näidise väärtus on lõpmatu list $1 : 2 : 3 : \ldots$, siis `x` väärtus on 1 ja `xs` väärtus lõpmatu list $2 : 3 : \ldots$. Kuid kui näidise `x : xs` väärtus on tühi list, siis näidis ei sobitu.

Haskellis teeb asja keerulisemaks asjaolu, et väärtus, millega näidise sobitumist tuleb kontrollida, on üldjuhul välja arvutamata. Selle asemel on mingi avaldis ning näidise sobitumise kontrollimisel väärtustatakse seda avaldist ainult niikaugele, et selguks, kas tema väärtuse struktuur vastab näidise omale sedavõrd, et saab rääkida näidise elementaarosadele vastavatest väärtuse osadest. Teisi sõnu, avaldist teisendatakse niikaua, kuni tulemus kas vastab näidise struktuurile kuni elementaarosadeni, mispuhul näidis loetakse sobituvaks ja tema elementaarosadega seotakse vastavad alamavaldised, või võtab näidise struktuurile vastukäiva kuju, mida edasiste teisenduste käigus enam muuta ei saa — siis loetakse näidis mittesobituvaks. Seega isegi kontrolli lõppedes ei ole enamasti väärtus teada. Näiteks võiks näidise `x : xs` väärtust $1:2:3:\ldots$ reaalselt esindada avaldis `[1 ..]`. Ka niisugusel juhul tehakse näidise sobitumine kindlaks, kuigi väärtuse väljaarvutamine on võimatu.

Protsessi, millesse on haaratud üks näidis ja avaldis ning mis hõlmab näidise sobitumise kontrolli avaldise väärtusega ja positiivse tulemuse korral näidise iga elementaarosa jaoks avaldise leidmise, millega ta väärtuselt võrdub, nimetatakse näidise **sobitamiseks** (ingl *match*).

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

Lihtsad deklaratsioonid

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

$$\mathfrak{p} \texttt{ = } \mathfrak{e}. \tag{4}$$

Kõige abstraktsemalt on deklaratsiooni (4) 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.

Kujul (4) oli ka meie esimene kirjutatud deklaratsioon (1). Seal oli vasakuks pooleks muutuja `main` ja paremaks pooleks avaldis `print (2 + (-3))`. Muutuja ongi üks võimalik näidise alaliik, nagu ülal juba ilmnes. Näiteks deklaratsioon

```
base
  = 10
```
(5)

defineerib muutuja `base` väärtuseks 10.

Ülesandeid

94. Kirjutada deklaratsioon (5) oma Haskell-faili. Testida defineeritud muutujat interaktiivses interpretaatoris.

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

Teine põhiline näidiseliik on näidis, mis on konstruktori abil koostatud osadest, mis on omakorda näidised. Seni vaatlesime kaht sellist näidist: `(a , b)`, mis on koostatud muutujatest `a` ja `b` paarikonstruktori abil, ja `x : xs`, mis on koostatud muutujatest `x` ja `xs` listikonstruktori `:` abil. Siia kuuluvad ka näidised, mis koosnevad paljast konstruktorist, nagu näiteks `True`, `False`, `Nothing`.

Näidistest konstruktoriga uue näidise koostamine käib sarnaselt avaldistest konstruktoriga uue avaldise koostamisega. Muuhulgas infiksoperaatorite prioriteedid ja assotsiatiivsus on samad mis avaldistes, samuti on võimalik kasutada muutujate infiks- ja üldkuju. Näiteks näidis `x : xs` on samaväärne näidisega `(:) x xs`. Kuid on ka oluline erinevus: konstruktoriga koostatud näidis ei tohi olla funktsioonitüüpi. Selle kohta öeldakse, et konstruktor peab näidises olema täisargumenteeritud. Näiteks näidise `(:) x` lükkab Haskell tagasi, sest ta on funktsioonitüüpi, ta vajab veel üht argumenti.

Konstruktoriga koostatud näidise elementaarosad on üldjuhul parajasti konstruktori argumentide elementaarosad. Kui konstruktoril argumendid puuduvad, siis puuduvad näidisel ka elementaarosad ja sobitumine sõltub ainult sellest, kas väärtus vastab konstruktorile. Näiteks näidis `True` sobitub väärtusega True, kuid ei sobitu väärtusega False.

Kuna konstruktor sisuliselt moodustab oma argumentidest andmestruktuuri, siis tema abil ehitatud väärtuse järgi on konstruktori argumentide väärtused alati üheselt määratud. Seega küsimuse konstruktoriga ehitatud näidise sobitumisest oma väärtusega saab lahendada järgnevalt. Kui väärtus ei ole ehitatud sama konstruktoriga, siis näidis ei sobitu temaga, sest struktuur ei klapi. Kui väärtus on ehitatud sama konstruktoriga, siis sobitub kogu näidis väärtusega parajasti siis, kui kõik konstruktori argumentnäidised sobituvad väärtuse vastava osaga, ning sealt saame ka elementaarosade väärtused.

Lihtne näide deklaratsioonist kujul (4), mille vasak pool on konstruktoriga ehitatud näidis, on

```
(x , y)
  = (5 , 0.2).
```
(6)

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

Deklaratsioon

```
p : ps
  = 1 : 3 : 4 : []
```
(7)

annab muutujale `p` väärtuseks 1 ja muutujale `ps` väärtuseks $3 : 4 : []$. Soovi korral võime rohkem elemente algusest muutujatega välja tuua, näiteks deklaratsioon

```
p : q : qs
  = 1 : 3 : 4 : []
```
(8)

defineerib `p` väärtuseks 1, `q` väärtuseks 3 ja `qs` väärtuseks $4 : []$.

Muidugi saame deklaratsioonid (7), (8) samaväärselt ümber kirjutada, kasutades paremates pooltes listide erisüntaksit. Saame vastavalt

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

p : q : qs
  = [1, 3, 4]·
```

Kuna stringid on sümbolite listid, on korrektne ka näiteks deklaratsioon

```
c : cs
  = "Hei!"'
```
(9)

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

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 stringkonstandid, samuti sümbol- ja arvkonstandid.

Ülesandeid

96. Millised näidistest

    ```
    (:), (:) x xs xss, (x : xs) : xss, [], [1],
    Just, Just a, Just sin, Just (+), Just Nothing
    ```

    on korrektsed?

97. Defineerida ühe deklaratsiooniga kaks muutujat, mille väärtusteks on vastavalt Teie ees- ja perekonnanimi stringina. 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.

98. Kirjutada oma moodulisse deklaratsioon, mis defineerib muutujate `q` ja `r` väärtusteks 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.

99. Modifitseerida deklaratsiooni (9) paremat poolt selliselt, et deklaratsiooni tulemusel saaks `cs` väärtuseks tühi string.

100. Kas saab kirjutada deklaratsiooni (7) vasaku poole samaväärselt ümber elementide loetelu kujul?

Näidise moodustamisel konstruktoriga viitavad konstruktori argumentnäidised struktuuri omavahel lõikumatutele osadele. Näiteks näidises `(x , y)` viitavad `x` ja `y` paari erinevatele komponentidele ja need ei lõiku omavahel. Haskell lubab ka olukorda, kus näidise muutujate väärtustest üks on teise osa.

Seda võimaldab nn **ehknäidis** (ingl *as-pattern*), mis konstrueeritakse, ühendades mingi muutuja ja mingi näidise infiksselt sümboliga `@` . See sümbol tähendab põhimõtteliselt näidiste võrdust; st kui näidise `x@ p` väärtus on x, siis on `x` väärtus x ja `p` väärtus x. Tulemusena on näidise `p` muutujate väärtused muutuja `x` väärtuse osad. Sümboli valik tuleb ingliskeelsest sõnast "*as*". Eesti keeles võib näidist kujul `x@ p` lugeda "`x` ehk `p`".

Ehknäidise `x@ p` elementaarosad on parajasti näidise `p` elementaarosad ja `x`.

Näiteks võiksime täiendada deklaratsiooni (6) vasakut poolt selliselt, et lisaks muutujate `x` ja `y` väärtuse määramisele kirjutataks kogu paari väärtus muutujasse `z`:

```
z@ (x , y)
  = (5 , 0.2)
```
. (10)

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"
```
. (11)

Kuna `@` on kõigist infikssetest seostest tugevaim, siis on vasaku poole näidis sama mis `t : (ts@ (u : us))`. Seega deklaratsiooni (11) tulemusena saab muutuja `t` väärtuseks stähe, näidis `ts@ (u : us)` aga stringi "tring". Järelikult nii muutuja `ts` kui näidise `u : us` väärtuseks saab "tring". Siit tulenevalt saab muutuja `u` väärtuseks t-täht ja muutuja `us` väärtuseks sõna "ring".

Ülesandeid

101. 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.

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

Veel üks näidiseliik on **jokker** (ingl *wildcard*), mis kirjutatakse üksiku alakriipsuna. Jokker on elementaarosadeta näidis, mis sobitub iga väärtusega. Jokkerit võib lugeda väljendiga "ükskõik mis". Sisuliselt tähistab jokker ebaolulist väärtust, see väärtus unustatakse ära, kuna ühtki tema osa ei seota.

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 (10), mis defineeris muutujad `x`, `y`, `z`. Oletame, et meil on vajalik kasutada muutujaid `x` väärtusega 5 ja `z` väärtusega $(5, 0.2)$, kuid `y` väärtusega 0.2 tarvis ei lähe. Siis võib deklaratsioonis (10) `y` asendada jokkeriga, kirjutades

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

Mittekasutatavate muutujate asendamine jokkeriga hoiab kokku arvuti mälu ja säästab ka koodi uurivat inimest nende muutujate tähenduse meelespidamisest.

Ülesandeid

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

Viimane liik näidiseid, mida me käsitleme, on nn **laisad** (*irrefutable*) näidised. Laisk näidis saadakse suvalisest näidisest prefiksse tildega, st kui `p` on näidis, siis `~p` on laisk näidis.

Näidise väärtust tilde ei muuda: `p` väärtus võrdub `~p` väärtusega. Laisk näidis on aga üks võimalus suuremat näidist "musta kasti panna" ehk elementaarseks kuulutada: kui `p` võib omada kuitahes keerulist struktuuri, siis näidisel `~p` on parajasti üks elementaarosa ja see on `p`.

Laisk näidis sobitub iga väärtusega, sest sõltumata tema väärtusest seotakse see väärtus 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 : [].
```
(12)

Siin näidis ei sobitu, 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 : []   .
```
(13)

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

Laiska näidist kasutatakse reaalselt selleks, et ennetada täitmisaegset viga. Näiteks deklaratsioonist (12) muutuja `p` väärtust küsides tekib näidisesobituse viga, kuid deklaratsiooni (13) puhul seda ei teki. Deklaratsioon (13) annab vea alles siis, kui küsida muutuja `q` või muutuja `qs` väärtust. Muutuja `p` väärtuse erinevust deklaratsioonide (12) ja (13) korral saab kergesti kontrollida interaktiivses keskkonnas.

Sellega oleme tutvunud põhiliste Haskelli näidistega ja anname nüüd üldise kirjelduse sellest, mis täpsemalt toimub, kui eksisteerib deklaratsioon kujul (4).

Kuni ühegi selle 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 (4) 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 otse näha. Mittesobitumise korral lõpetatakse töö täitmisaegse veaga. Sobitumise korral seotakse näidise $\mathfrak{p}$ kõik elementaarosad avaldise $\overline{\mathfrak{e}}$ vastavate alamavaldistega (need võivad veel olla väärtustamata). Kui $\mathfrak{x}$ oli üks elementaarosadest, siis on leitud avaldis, millest võib edasi hakata tema väärtust arvutama. Kui $\mathfrak{x}$ polnud elementaarosa, siis otsitakse välja elementaarosa $\mathfrak{p}'$, kus $\mathfrak{x}$ esineb; olgu $\mathfrak{e}'$ temaga eelmisel etapil seotud avaldis. Edasi toimitakse analoogiliselt sellega, nagu oleks kirjutatud deklaratsioon kujul (4), kus vasak ja parem pool on vastavalt $\mathfrak{p}'$ ja $\mathfrak{e}'$.

Illustreerime seda protsessi deklaratsiooni (6) 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 (6) 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 (6) enam kasutama.

Põhimõtteliselt samamoodi käivad asjad deklaratsiooni (8) 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 nüüd 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 olnud vaja muutuja `q` väärtust, siis tulnuks uurida veel näidise `q : qs` sobitumist avaldise `3 : 4 : []` väärtusega, millest `q` seotuks avaldisega `3`.

Ülesandeid

104. Kirjutada deklaratsioon (12) oma Haskell-faili ja tekitada interaktiivses keskkonnas näidisesobituse viga.

105. 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.

106. Demonstreerida arvutis, et muutuja, jokkeri ja laisa näidisega sobitub ka bottom.

107. 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.

Muutujate definitsioonidele võib lisada **tüübisignatuure** (ingl *type signature*) — deklaratsioone, mis näitavad defineeritava muutuja tüüpi.

Süntaktiliselt on tüübisignatuur sarnane annoteeritud avaldisega; tüübisignatuuri üldine kuju on

$$x_1, \ldots, x_l :: t,$$

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 $\mathfrak{t}$.

Haskell nõuab, et tüübisignatuuri vasakus pooles olevad muutujad oleksid ka defineeritud. Signatuur ja definitsioon on eraldi asjad, muutuja tüübisignatuur ei saa olla sama muutuja definitsiooni sees. Signatuuri võib võtta kui definitsiooni juurde kuuluvat lisadeklaratsiooni. 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 (5), mis defineerib muutuja `base` väärtuseks 10, võib lisada tüübisignatuuri

```
base
  :: (Num a) => a·
```
(14)

Kui tahame kitsendada muutuja `base` tüübiks `Integer`, võib signatuuri (14) asemel kirjutada

```
base
  :: Integer·
```

Enamasti ongi 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 lisamine on mõttekas igal juhul, kuna see hõlbustab koodist arusaamist.

Tüübiteema juurde märgime, et kuigi seni oleme kasutanud vaid andmenäidiseid, on näidistest mõtet rääkida ka tüüpide tasemel. Ka globaalseid tüübimuutujaid saab defineerida deklaratsiooniga kujul (4), kui ette lisada võtmesõna **`type`**.

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

```
type Rida
  = [Integer]·
```

Pärast seda on võimalik kirjutada avaldisi nagu `[1 .. ] :: Rida` jms.

Siiski on tüübimuutujate defineerimisel tugevad kitsendused võrreldes andmemuutujatega. Tüübinäidis tohib sellises deklaratsioonis olla ainult maksimaalselt lihtne ehk üksik muutuja ning tüübiavaldis peab olema kontekstita.

Ülesandeid

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

Lambdaavaldised

Teine tüüpiline koht, kus näidised figureerivad, on funktsioonide definitsioonid, seal täidavad näidised formaalsete argumentide aset.

Vaatame kõige lihtsamat sellistest süntaktilistest konstruktsioonidest — lambdaavaldist (ingl *lambda-expression*), mis määravad funktsioonitüüpi väärtuse ilma talle nime andmata.

Lambdaavaldis on kujul

```
\ p -> e,
```

kus `p` on näidis ja `e` avaldis. Noole moodi kombinatsioon `->` nagu varem võrdusmärkki jagab kirjutise vasakuks ja paremaks pooleks. Sümbolit `\` loetakse "lambda". Kogu selle avaldise väärtuseks on funktsioon, mis igal oma argumendil x annab tulemuseks avaldise `e` väärtuse eeldusel, et näidise `p` väärtus on x ning iga muutuja, mis esineb nii näidises `p` kui ka avaldises `e`, on neis sama väärtusega. Kui näidis argumendi väärtusega ei sobitu, siis on funktsiooni väärtus sel argumendil $\perp$.

Näiteks avaldis `\ x -> x * x` väärtuseks on funktsioon, mis igal oma argumendil x annab väärtuseks avaldise `x * x` väärtuse eeldusel, et `x` väärtus on x, ehk väärtuse 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` väärtuseks on funktsioon, mis igal oma argumendil $(a\,, b)$ annab väärtuseks 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.

Lambdaavaldise vasakus pooles esinevad muutujad on lokaalsed, nähtavad ainult selle lambdaavaldise piires. Näiteks `x` näiteavaldises `\ x -> x * x` on lokaalne muutuja, tema nähtavus piirdub ainult selle lambdaavaldisega. Põhimõtteliselt tohivad lokaalsete muutujate nimed langeda kokku globaalsete muutujate nimedega, sellisel puhul varjutatakse vastavad globaalsed muutujad lokaalse muutuja nähtavuspiirkonnas, nad pole seal kasutatavad. Näiteks avaldise `\ sin -> sin * sin` väärtus on seesama ruutfunktsioon, kuigi `sin` on eeldefineeritud muutuja.

Pangem veel tähele, et lambdaavaldiste puhul on väärtuste liikumise suund vastupidine varemvaadeldud deklaratsioonidega võrreldes. Kui deklaratsioonis kujul (4) liigub väärtus avaldisest näidisesse, siis lambdaavaldises liiguvad muutujate väärtused näidisest avaldisse, näidisesse liigub aga argumendi väärtus.

Lambdaavaldist saab väärtustada ainult koos argumendiga. Avaldise `(\ p -> e) a` väärtustamisel väärtustatakse kõigepealt argumenti `a` parajasti niipalju, et ilmneks näidise `p` sobitumine või mittesobitumine tema väärtusega. Sobitumise korral seotakse näidise elementaarosad avaldi-

se vastavate osadega, kogu avaldis aga kirjutatakse ümber lihtsalt avaldiseks ϵ. Mittesobitumise korral antakse täitmisaegne viga.

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 väätustata. Näidisesobitus seob muutuja `x` avaldisega `1 + 2`. Algne avaldis aga kirjutatakse ümber avaldiseks `x * x`. Nüüd on juba näha, et edasine väärtustamine annab tulemuseks `9`, aga väärtustamise selles osas ei mängi lambdaavaldis enam mingit rolli.

Keerulisema näitena vaatleme avaldise

```
(\ ~(x : xs) -> const 5 x) []
```
(15)

väärtustamist. Kuna näidis `~(x : xs)` on laisk, siis ta sobitub tühja listiga. Näidis `x : xs` seotakse avaldisega `[]`. Avaldis (15) kirjutatakse ümber avaldiseks `const 5 x`. Nüüd kuna `const 5` on laisk, siis muutuja `x` väärtust vaja ei ole ja väärtustamise lõpptulemuseks on `5`.

Ülesandeid

109. Testida ruutfunktsiooni interaktiivse interpretaatori käsurealt.

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

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

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

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

114. Kirjutada võimalikult lühike ja võimalikult vähe muutujaid kasutav 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.

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

116. Kirjeldada detailselt avaldise `(\ (x , _) -> x) (1 + 1 , 2 - 5)` väärtustusprotsessi.

117. Asendada avaldises (15) muutuja `const` sellise muutujaga, et tulemuseks oleva avaldise väärtustamine lõpetab täitmisaegse veaga.

118. Kui avaldises (15) kaotada tilde, mis on tulemuseks oleva avaldise väärtus?

Lambdaavaldisega on võimalik üles kirjutada ka *curried*-kujul funktsioone. Selleks peab vaid paremas pooles olema funktsioonitüüpi avaldis. See võib olla omakorda lambdaavaldis.

Näiteks *curried*-kujul funktsiooni, mis leiab kahe arvu aritmeetilise keskmise, määrab avaldis

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

Analoogne funktsioon kolme argumendi korral oleks

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

Curried-kujul funktsioonide jaoks on olemas ka kirjutamist lihtsustav erisüntaks, mis võimaldab korduvad lambdad ja nooled kaotada, jättes alles ainult esimese lambda ja viimase noole ning kirjutades kõik argumendid lihtsalt üksteise järele neid vajadusel tühisümbolitega eraldades. Kahe ja kolme arvu aritmeetilist keskmist arvutavad funktsioonid oleks selle süntaksiga vastavalt

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

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

Selle erisüntaksiga lisandub nõue, et sama lambda alla kogutud näidiste reas ei tohi ükski muutuja esineda korduvalt — muidu kehtib see nõue vaid igas näidises eraldi.

Ülesandeid

119. Kirjutada lambdaavaldis, mille väärtuseks on funktsiooni `snd` *curried*-kuju.

120. Kirjutada ja testida interaktiivse interpretaatori käsurealt ülesandes 111 kirjutatud funktsiooni *curried*-kuju.

121. Kirjutada lambdaavaldis, mille väärtuseks on *curried*-kujul 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.

Pannes deklaratsiooni (4) paremaks pooleks lambdaavaldise, saame defineerida muutuja väärtuseks endakoostatud funktsioone. Näiteks võime defineerida ruutfunktsiooni muutuja `sqr` väärtuseks deklaratsiooniga

```
sqr
  = \ x -> x * x'
```
(16)

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

```
am2
  = \ a b -> (a + b) / 2'
```
(17)

```
am3
  = \ a b c -> (a + b + c) / 3.
```
(18)

Deklaratsioonide (17) ja (18) olemasolul on muutujad `am2` ja `am3` oma uues tähenduses kasutatavad nii prefiks- kui infikskujul. Näiteks avaldiste `am2 3 5` ja `3 'am2' 5` väärtus on 4, avaldiste `am3 1 6 8` ja `(1 'am3' 6) 8` väärtus aga 5.

Kui on ette näha, et uut muutujat hakatakse oma argumentidele rakendama peamiselt infiksselt, on mõistlik muutujanimi koostada sümbolitest reas (2). Näiteks deklaratsioon

```
(//)
  = \ a x -> fromIntegral a + 1 / x
```
(19)

annab muutujale `//` väärtuseks *curried*-kujulise funktsiooni, mis võtab argumendiks täisarvu a ja murdarvu x ja annab tulemuseks arvu $a + \frac{1}{x}$.

Värskelt kasutusele võetud nime infikskuju prioriteet ja assotsiatiivsus võrduvad vaikeväärtustega, kui neid pole ümber defineeritud. Viimast saab teha **infiksdeklaratsiooniga**. Infiksdeklaratsiooniks kõlbab selline rida, mille esitavad interaktiivsed keskkonnad infiksoperaatorite prioriteedi-assotsiatiivsuse kohta kasutajale, kuid sama infiksdeklaratsiooniga saab sama prioriteeti ja assotsiatiivsust seada mitme nime jaoks.

Infiksdeklaratsiooni üldkujud on

infixa p $\oplus_1$, ... , $\oplus_l$,
infixa $\oplus_1$, ... , $\oplus_l$,

kus `a` on `l`, `r` või tühjus, mis tähendavad vastavalt vasak-, parem- ja mitteassotsiatiivsust, `p` on number `0`-st `9`-ni, mis tähendab prioriteeti, ja $\oplus_1, \ldots, \oplus_l$ on infiksoperaatorid, kusjuures $l > 0$. Sellise deklaratsiooni tulemusena muutub infiksoperaatorite $\oplus_1, \ldots, \oplus_l$ assotsiatiivsus vastavalt sellele, kuidas nad on deklaratsioonis märgitud. Kui prioriteet puudub, jääb jõusse vaikeprioriteet.

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

122. Lisada definitsioonile (19) infiksdeklaratsioon, mis annab infiksoperaatorile `//` jagamisega võrdse prioriteedi ja mõistliku assotsiatiivsuse.

Hargnemiskonstruktsioonid

Tutvume järgnevalt nelja süntaktilise konstruktsiooniga, mis lubavad erinevate juhtude läbivaatust. Kaks esimest neist moodustavad avaldise. Hoolimata sellest vaatleme neid kõiki uute muutujate definitsioonide koosseisus, sest ühelt poolt on nende konstruktsioonidega moodustatud süntaktilised objektid enamasti liiga pikad, et neid interaktiivses keskkonnas käsurealt sisestada, ja teiselt poolt on hargnemine mõttekas ainult siis, kui ta toimub mingi tundmatu väärtuse põhjal, milleks tüüpiliselt on defineeritava funktsioonitüüpi muutuja argument.

Tingimusavaldis

Lihtsaim hargnemisega konstruktsioon on tingimusavaldis kujul

$$\texttt{if } \mathfrak{b} \texttt{ then } \mathfrak{e}_1 \texttt{ else } \mathfrak{e}_2, \tag{20}$$

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 (20) väärtus võrdub $\mathfrak{e}_1$ väärtusega; kui $\mathfrak{b}$ väärtus on False, siis avaldise (20) väärtus võrdub $\mathfrak{e}_2$ väärtusega; kui $\mathfrak{b}$ väärtus on $\perp$, siis avaldise (20) väärtus on $\perp$.

Näiteks deklaratsioon

```
gm2
  = \ x y
      -> if x > 0 && y > 0                                    (21)
           then sqrt (x * y)
           else error "gm2: argumendid olgu positiivsed"
```

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

Deklaratsioon

```
tõeväärtusArvuks
  = \ x -> if x then 1 else 0                                 (22)
```

defineerib muutuja `tõeväärtusArvuks` väärtuseks funktsiooni, mis teisendab tõeväärtusi arvulisele kujule.

Väärtustamaks avaldist (20), väärtustatakse kõigepealt avaldis $\mathfrak{b}$, niikaua kui tuleb välja `True` või `False`. Vastavalt sellele jätkatakse kas $\mathfrak{e}_1$ või $\mathfrak{e}_2$ väärtustamisega.

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. C-s ja Javas on Haskelli tingimusavaldise analoogiks pigem konstruktsioon *<tingimus>* `?` *<avaldis>* `:` *<avaldis>* kui *if*-lause. Kuna Haskellis saab avaldise väärtuseks olla protseduur, saab Haskelli tingimusavaldis muidugi tähistada ka imperatiivset *if*-lauset.

Kuna tegemist on avaldisega, on konstruktsiooni (20) kasutamine niisama vaba kui avaldise kasutamine üldse. Näiteks võib ta asetada teise avaldise sisse. Defineerimaks muutuja `valiKääne` väärtuseks funktsiooni, mis võtab argumentideks arvu ja kaks stringi ning annab välja esimese stringi, kui arv võrdub 1-ga, ja teise stringi 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”.

Valikuavaldis

Valikuavaldise puhul tehakse valik harude vahel, sobitades erinevaid näidiseid ühe kindla väärtusega. Valikuavaldis koosneb päisest kujul **`case`** $\mathfrak{e}$ **`of`**, kus $\mathfrak{e}$ on avaldis, mille vastu näidiseid sobitatakse, ja juhtude loendist. Iga juht sisaldab näidise koos vastava parema poolega, mis tuleb valida selle näidise sobitumise korral.

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

$$\begin{array}{l}\textbf{case}\ \mathfrak{a}\ \textbf{of}\\ \quad \mathfrak{p}_1\ \textbf{->}\ \mathfrak{e}_1\\ \quad \ldots\ldots\ldots\\ \quad \mathfrak{p}_l\ \textbf{->}\ \mathfrak{e}_l\end{array}\ , \tag{23}$$

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 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 sama 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 (23) väärtuseks on $\mathfrak{e}_i$ väärtus eeldusel, et 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 $\bot$ ja näidis $\mathfrak{p}_1$ sobitub sellega. Kui ükski näidistest $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ ei sobitu avaldise $\mathfrak{a}$ väärtusega, või $\mathfrak{a}$ väärtus on $\bot$ ja $\mathfrak{p}_1$ ei sobitu $\bot$-ga, siis on avaldise (23) väärtus $\bot$.

Näiteks deklaratsioon

```
üksLõppu
  = \ xs
      -> case xs of
           z : zs                                    (24)
             -> zs ++ [z]

           _
             -> []
```

defineerib `üksLõppu` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see on mittetühi, siis annab väärtuseks 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 : _                                     (25)
             -> z : xs

           _
             -> []
```

defineerib `kordaEsimest` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see on mittetühi, siis annab väärtuseks 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"                               (26)
           1
             -> "Üks"

           _
             -> "Mitu"
```

defineerib `primLoenda` väärtuseks funktsiooni, mis võtab argumendiks listi: kui see on lõplik, siis annab väärtuseks eestikeelse teksti, mis ütleb, kas listis oli elemente null, üks või mitu; kui list on lõpmatu, siis on väärtuseks $\bot$.

Valikuavaldise (23) väärtustamine käib järgnevalt. Avaldis $\mathfrak{a}$ väärtustatakse parajasti niikaugele, et selguks näidise $\mathfrak{p}_1$ sobitumine või mittesobitumine tema väärtusega. Kui ta sobitub, siis seotakse $\mathfrak{p}_1$ elementaarosad ja kirjutatakse avaldis (23) ümber avaldiseks $\mathfrak{e}_1$. Kui ta ei sobitu, siis $\mathfrak{p}_1$ elementaarosi ei seota. Kui seejuures $\mathfrak{a}$ väärtus on normaalne, siis heidetakse esimene variant kõrvale ja jätkatakse järjest samamoodi ülejäänud variantidega. Kui variandid saavad otsa, siis antakse täitmisaegne viga.

Illustreerime seda skeemi deklaratsiooni (26) peal. Oletame, et on vaja väärtustada avaldis

```
primLoenda (take 1 "ABC").
```
(27)

Kõigepealt kirjutatakse `primLoenda` deklaratsioonist (26) ümber deklaratsiooni parema poolega, milleks on lambdaavaldis. Vastavalt lambdaavaldise väärtustusskeemile tuleb nüüd väärtustada argumenti `take 1 "ABC"` seni, kuni selgub näidise `xs` sobitumine tema väärtusega. Kuna muutuja sobitub triviaalselt iga väärtusega, 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 väärtustatakse teda seni, kuni selgub esimese näidise `0` sobitumine või mittesobitumine tema väärtusega. See ei selgu enne, kui `length xs` on väärtustatud lõpuni. Kuna `xs` on seotud avaldisega `take 1 "ABC"`, mille väärtus on string “A”, siis `length xs` väärtustamine annab tulemuseks `1`. Kuna `0` ei sobitu väärtusega 1, kuid 1 on siiski normaalne väärtus, siis proovitakse edasi järgmist varianti. Seal on näidis `1`, mis sobitub väärtusega 1. Seetõttu kirjutatakse kogu avaldis ümber stringikonstandiks `"Üks"`. See on ka lõppväärtus.

Mõeldes väärtustamise käigu peale, saab selgeks, et `primLoenda` defineerimine deklaratsiooniga (26) on tegelikult väga ebaõnnestunud, sest valikuavaldise esimese näidise sobitamiseks tuleb argumentlisti pikkus välja arvutada, see on aga lineaarse keerukusega töö listi pikkuse suhtes. 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" .
           [_]
             -> "Üks"
           _
             -> "Mitu"
```
(28)

Väärtustades selle definitsiooni järgi, selgub esimese näidise sobitumine või mittesobitumine `xs` väärtusega juba siis, kui `xs` väärtuse kujust on välimine konstruktor käes. 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 välimine konstruktor kätte saada. Seetõttu annab see kood mõistliku tulemuse alati, kui `xs` väärtuseks on lõplik või lõpmatu list või selline osaline list, kus on vähemalt kaks elementi.

Teine asi, mida avaldise (27) deklaratsiooni (26) järgi väärtustamise käik selgitab, on põhjus, miks bottomiga sobitatakse ainult valikuavaldise esimest näidist. Kui avaldises (23) on $\mathfrak{a}$ väärtus $\bot$ sellepärast, et tema väärtustamine lõpetab töö täitmisaegse veaga, siis oleks ju võimalik see viga unustada ja proovida ka järgmisi näidiseid, millest mõni võib sobituda. Samas kui $\mathfrak{a}$ väärtus on $\bot$ sellepärast, et tema väärtustamine jääb lõpmatusse tsüklisse ja ei anna iialgi ühtki konstruktorit välja, mille alusel näidise sobituvust kontrollida — nii juhtub, kui `primLoenda` argumendi väärtus on lõpmatu —, 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.

Kõik tingimusavaldised on võimalik asendada valikuavaldisega: kehtib samaväärsus

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

Mõlemal juhul on väärtuseks $\mathfrak{e}_1$ väärtus, kui $\mathfrak{b}$ väärtus on True, ja $\mathfrak{e}_2$ väärtus, kui $\mathfrak{b}$ väärtus on False; kui $\mathfrak{b}$ väärtus on $\bot$, siis on mõlema avaldise väärtus $\bot$.

Näiteks võiks definitsiooni (21) asendada deklaratsiooniga

```
gm2
  = \ x y
      -> case x > 0 && y > 0 of
           True                                                    (29)
             -> sqrt (x * y)
           _
             -> error "gm2: argumendid olgu positiivsed"
```

Mitme tingimuse järgi hargnemist, mida vaid tingimusavaldistega kodeerides tuleks tingimusavaldisi üksteise sisse panna, saab valikuavaldisega mõnikord ühe hargnemisega teha.

Olgu meil näiteks tarvis muutuja `absSuurem` väärtuseks defineerida *curried*-kujuline funktsioon, mis võtab argumentideks kaks täisarvu: 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. Kolmeks hargnemine võrdlustulemuse põhjal on tüüpiline olukord, kus kasutatakse valikuavaldist ja tüüpi Ordering. Moodulist Prelude saab muutuja `compare`, mille väärtus on *curried*-kujuline funktsioon, mis võtab argumentideks kaks objekti samast klassi Ord tüübist ja annab tulemuseks nende võrdlustulemuse tüüpi Ordering kuuluva väärtusena. Neid väärtusi esitavad konstruktorid `LT`, `EQ`, `GT`. Nii saame nõutud muutuja defineerida deklaratsiooniga

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

Kasutades võrdlemist muutujaga `compare`, saavutame mitte ainult ühe hargnemisega koodi, vaid ka ühe võrdlusega arvutuse, mis on oluline efektiivsuse kaalutlustel.

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

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

Mõlemal juhul on väärtuseks avaldise `e` väärtus eeldusel, et näidise `p` väärtus võrdub `a` väärtusega ning näidises `p` esinevad muutujad on `p` ja `e` piires sama väärtusega.

Ülesandeid

123. 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.

124. 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. Arvutus peab käima konstantse keerukusega listi pikkuse suhtes.

125. 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.

126. Kasutades tüüpi Ordering, defineerida muutuja `võrdle` väärtuseks *curried*-kujul funktsioon, mis võtab ükshaaval argumendiks täisarvud x, y ja annab väärtuseks stringi "Võrdsed", "Järjestikused" või "Kauged" vastavalt sellele, kas $x = y$, x ja y on järjestikused täisarvud (ükskõik kummas järjekorras) või ei kehti kumbki neist tingimustest.

127. Kirjeldada detailselt avaldise `üksLõppu "ABC"` väärtustusprotsessi, kus `üksLõppu` on defineeritud deklaratsiooniga (24).

128. Kirjeldada detailselt avaldise `kordaEsimest [3, 5, 15]` väärtustusprotsessi, kus `kordaEsimest` on defineeritud deklaratsiooniga (25).

129. Kirjutada avaldis (22) samaväärselt ümber valikuavaldisena.

130. Kirjutada avaldis (15) samaväärselt ümber valikuavaldisena.

Deklaratsioonisüsteemid

Funktsioonide defineerimiseks muutujate väärtuseks võib kasutada deklaratsioone, mis erinevad senivaadelduist selle poolest, et vasak pool koosneb mitmest näidisest. Selline nn **funktsionaalne** vasak pool on kujul

$$\mathfrak{f}\ \ \mathfrak{p}_1\ \ \ldots\ \ \mathfrak{p}_l, \tag{30}$$

kus $\mathfrak{f}$ on muutuja ning $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ on näidised, kusjuures $l > 0$.

Deklaratsioon, mille vasak pool on kujul (30), defineerib muutujat $\mathfrak{f}$, tema väärtuseks saab teatav funktsioon. Näidised $\mathfrak{p}_1, \ldots, \mathfrak{p}_l$ märgivad funktsiooni argumente. Kuid see funktsioon ei pruugi olla määratud ainuüksi selle deklaratsiooniga; võib leiduda veel deklaratsioone, mille vasak pool on samal kujul ja esimene näidis on $\mathfrak{f}$. Sellised deklaratsioonid moodustavad kokku **deklaratsioonisüsteemi** ja defineeritava muutuja $\mathfrak{f}$ väärtuse määrab süsteem koos.

Süsteemi moodustavad deklaratsioonid peavad asetsema koodis järjest, nende vasakud pooled peavad sisaldama ühepalju näidiseid ning ükski muutuja ei tohi esineda sama deklaratsiooni argumendinäidiste reas korduvalt. Tüübikorrektsuseks peavad erinevate deklaratsioonide vastavad näidised olema sama tüüpi. Paremad pooled võivad olla sellised, nagu seni vaadeldud; siis on süsteem kujul

$$\begin{array}{l} \mathfrak{f}\ \mathfrak{p}_{1,1}\ \ \ldots\ \ \mathfrak{p}_{1,l} \\ \quad \texttt{=}\ \mathfrak{e}_1 \\ \texttt{..............} \\ \mathfrak{f}\ \mathfrak{p}_{k,1}\ \ \ldots\ \ \mathfrak{p}_{k,l} \\ \quad \texttt{=}\ \mathfrak{e}_k \end{array} \tag{31}$$

Deklaratsioonisüsteemi sisuks on juhtude läbivaatus, iga deklaratsioon märgib üht juhtu. Muutuja $\mathfrak{f}$ väärtuseks saab funktsioon, mis võtab ükshaaval argumendiks objektid $x_1, \ldots, x_l$. Kui i on vähim arv, mille korral i-nda deklaratsiooni kõik näidised sobituvad nende argumentidega vastavalt ja ükski argument pole $\perp$, siis annab funktsioon tulemuseks $\mathfrak{e}_i$ väärtuse eeldusel, et 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 $\perp$, kuid esimese i deklaratsiooni selle argumendi näidised sobituvad temaga. Ülejäänud juhtudel on funktsiooni väärtuseks neil argumentidel $\perp$.

Kui deklaratsioonide arv süsteemis on 1 ehk süsteem koosneb ühest deklaratsioonist

$$\begin{array}{l} \mathfrak{f}\ \mathfrak{p}_1\ \ \ldots\ \ \mathfrak{p}_l \\ \quad \texttt{=}\ \mathfrak{e} \end{array}, \tag{32}$$

siis on see deklaratsioon samaväärselt ümber kirjutatav deklaratsiooniga

$$\mathfrak{f}\ \texttt{= \textbackslash}\ \mathfrak{p}_1\ \ \ldots\ \ \mathfrak{p}_l\ \texttt{->}\ \mathfrak{e},$$

kus vasak pool pole funktsionaalne.

Funktsionaalse vasaku poolega deklaratsioonidega saame pisut lühemalt ja loetavamalt ümber kirjutada kõik senised deklaratsioonid, mis defineerivad muutujate väärtuseks funktsioone.

Näiteks ruutfunktsioon sai muutuja `sqr` väärtuseks defineeritud deklaratsiooniga (16); uues süntaksis teeb sama deklaratsioon

```
sqr x
  = x * x.
```
(33)

Deklaratsioonid (17) ja (18) on samaväärselt ümber kirjutatavad vastavalt definitsioonideks

```
am2 x y
  = (x + y) / 2,

am3 x y z
  = (x + y + z) / 3.
```

Curried-kujul funktsioonide defineerimisel võib defineeritav muutuja olla ka infiksne. Niisuguse deklaratsiooni funktsionaalne vasak pool on kujul

$$(\mathfrak{p}_1 \ \oplus \ \mathfrak{p}_2)\ \mathfrak{p}_3 \ \ldots \ \mathfrak{p}_l,$$

kus $\oplus$ on defineeritava muutuja infikskuju ja $l \geqslant 2$. Seega muutujad `am2`, `am3` saame varasemaga samaväärselt defineerida ka vastavalt deklaratsioonidega

```
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.

Funktsionaalses vasakus pooles kehtivad kõik prioriteedid ja assotsiatiivsused nagu avaldisteski, kusjuures näidiste järjestkirjutamine on sama prioriteediga nagu avaldises funktsioonirakendamist tähistav järjestkirjutamine, st see on kõigist infiksoperaatoritest kõrgem.

Hargnemise võimaluste nägemiseks võib deklaratsioonisüsteemiga ümber kirjutada varemantud definitsioone, mille paremas pooles on valikuavaldis. Näiteks deklaratsiooniga (24) defineeritud muutuja `üksLõppu` defineerib samaväärselt süsteem

```
üksLõppu (z : zs)
  = zs ++ [z]
üksLõppu _
  = []
```
. (34)

Nii otsene ümberkirjutamine on võimalik siiski ainult juhul, kui avaldis valikuavaldise päises langeb kokku mõne argumendinäidisega ja argumendinäidise muutujad mujal samas tähenduses ei esine, nagu see on deklatsioonides (24) ja (28).

Deklaratsioonisüsteemi eelised valikuavaldise ees ilmnevad juhul, kui hargnemisotsuseid tehes tuleb arvestada korraga mitme argumendi väärtusi.

Näiteks süsteem

```
risti (x : _) (_ : ys)
  = x : ys
risti _       _
  = []
```
(35)

defineerib muutuja `risti` väärtuseks *curried*-kujul funktsiooni, mis võtab kaks argumentlisti: kui mõlemad on mittetühjad, siis annab välja esimese listi peast ja teise listi sabast koostatud

listi; 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 koodi täitmise ajal ning kood jookseks mõnevõrra aeglasemalt.

Ülesandeid

131. Otsida Hugsi teegist üles muutujate `fst`, `snd`, `head`, `tail`, `const` definitsioonid ja saada neist aru.

132. Defineerida uue meetodiga muutuja `erinevus` väärtuseks ülesandes 111 kirjeldatud funktsioon.

133. Võimalikult lühikese ja võimalikult vähe muutujaid kasutava deklaratsiooniga kujul (32) defineerida muutuja `imelik` väärtuseks ülesandes 114 kirjeldatud funktsioon.

134. Defineerida muutuja `takeLõpust` väärtuseks funktsioon, mis töötab nagu `take`, aga võtab elemente listi lõpust. Näiteks `takeLõpust 2 [3, 2, 4]` väärtustamine peab andma `[2, 4]`.

135. Defineerida muutuja `currErinevus` väärtuseks ülesandes 111 kirjeldatud funktsiooni *curried*-kuju, kasutades defineeritavat muutujat prefiksselt.

136. Modifitseerida ülesandes 135 kirjutatud definitsiooni nii, et defineeritav muutuja oleks kasutatud infiksselt.

137. Muuta ülesandes 135 defineeritud muutuja infikskuju prioriteet ja assotsiatiivsus sarnaseks plussi ja miinusega.

138. Lahendada ülesanded 123, 124, 125 deklaratsioonisüsteemiga.

139. Kirjutada muutuja `primLoenda` definitsioon (28) samaväärselt ümber deklaratsioonisüsteemiga.

140. Kirjutada muutuja `kordaEsimest` definitsioon (25) samaväärselt ümber mitme deklaratsiooni tehnikaga, kasutades samu muutujaid samas tähenduses.

141. Defineerida muutuja `pead` väärtuseks *curried*-kujul funktsioon, mis võtab argumentideks kaks listi ja annab väärtuseks listi, mille elementideks on kõik need objektid, mis esinevad esimese või teise listi esimese elemendina, igaüks täpselt üks kord.

142. Kirjutada oma Haskell-faili muutuja `risti` definitsioon (35) ja veenduda, et see töötab. Seejärel asendada teine deklaratsioon deklaratsiooniga

```
risti
  = \ _ _ -> []
```

Tutvuda veateatega, mis tekib.

143. Kas saame süsteemiga (35) samaväärse definitsiooni, kui viime korraga mõlema deklaratsiooni argumendid lambdaavaldisega paremasse poolde (nii nagu ülesandes 142 tehti teise deklaratsiooniga)?

Kuna tüüpidel on võimalikud ainult muutujanäidised (Haskelli laiendused aktsepteerivad ka jokkerit), ei ole tüübimuutujate definitsioonides hargnemine võimalik. Sellegipoolest 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.)

Valvurikonstruktsioon

Deklaratsiooni või valikuavaldise juhu paremaks pooleks võib lisaks taolistele, mida seni oleme kasutanud, olla **valvurikonstruktsioon**. Selle konstruktsiooni mõte on esitada üleminek sõltuvalt lisatingimustest.

Deklaratsiooni parema poolena näeb valvurikonstruktsioon välja kujul

```
  | g1 = e1
..........
  | gl = el
```

Vasakul pool olevaid objekte $\mathfrak{g}_i$ nimetatakse **valvuriteks** (ingl *guard*). Iga valvur on tõeväärtusetüüpi avaldis ehk tingimus. Vastavas paremas pooles on avaldis, mis tuleb valida juhul, kui see valvur on esimene, mis väärtustub tõeseks, ja varasemate väärtused on normaalsed.

Valikuavaldise juhu parema poolena näeb valvurikonstruktsioon välja samamoodi, ainult võrdusmärkide asemel on noolemärgid `->`.

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   .                                    (36)
    = "Nullilähedane"
  | x >= 1
    = "Suur"
```

Iga valvurini jõutakse ainult siis, kui ükski eelnev samas komplektis pole sobinud. Seega deklaratsiooni (36) teine valvur `x >= 0 && x < 1` tuleb kontrollimisele 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. 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 (36) 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             .                                    (37)
    = "Nullilähedane"
  | otherwise
    = "Suur"
```

Selle järgi arvutades tuleb teha maksimaalselt 2 võrdlust.

Valvurikonstruktsioon on eriti mugav juhul, kui tingimusi on palju. 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. Arvu täisosa leidmise funktsioon on väärtuseks mooduli Prelude

muutujal `floor`. Seega saame deklaratsiooni

```
arvuKlass x
  = case compare (floor x) 0 of
      LT
        -> "Negatiivne"
      EQ
        -> "Nullilähedane"
      _
        -> "Suur"
```

Valvurikonstruktsioonid ei ole avaldised, nad moodustavad omaette süntaktilise kategooria. Kui valvurikonstruktsiooni ükski valvur ei väärtustu tõeseks ja tegemist on deklaratsioonisüsteemi deklaratsiooni või valikuavaldise juhuga, siis loetakse deklaratsioon või juht mittesobivaks ja valitakse järgmine, justnagu vasaku poole mõne näidise sobitus oleks ebaõnnestunud. Niisiis, erinevalt senistest hargnemistest, ei anna siin kõigi harude mittesobivus automaatselt täitmisaegset viga, vaid hargnemiste puus ühe taseme võrra tagasi pöördumise.

Vaatleme valvurikonstruktsiooni sisaldavat deklaratsiooni

```
listiKlass (z : _)
  | z > 0
    = ":)"
  | z == 0
    = ":|"
listiKlass _
  = ":("
```

(38)

Mängime läbi muutuja `listiKlass` rakendamise mingile listitüüpi avaldisele ℓ definitsiooni (38) järgi. Kui ℓ 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 string ":)". Kui listi pea on null, siis esimene valvur väärtustub vääraks, teine aga tõeseks, mistõttu rakendamise tulemus on string ":|". 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 kirjutada siiski ka ühe deklaratsiooni ja ühel

tasemel hargnemisega. Sobib deklaratsioon

```
listiKlass xs@ ~(z : _)
  | null xs || z < 0
    = ":("
  | z > 0                .
    = ":)"
  | otherwise
    = ":|"
```
(39)

Vaatame, kuidas toimub `listiKlass` ɬ väärtustamine definitsiooni (39) korral. Alguses seotakse näidised `xs` ja `z : _` avaldisega ɬ, siis minnakse valvurikonstruktsiooni juurde. Kui ɬ väärtus on tühi, väärtustub esimese valvuri disjunktsiooni vasak pool ja seega ka kogu valvur tõeseks ilma disjunktsiooni teist poolt uurimata ning `listiKlass` ɬ 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 ɬ 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 ɬ. Kuna ɬ väärtus on mittetühi, siis see õnnestub ja `z` seotakse avaldisega, mille väärtus on listi pea. Edasi on juba selge — käitutakse vastavalt `z` väärtusele.

Ülesandeid

144. 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.

145. Lahendada ülesanne 141 valvurikonstruktsiooni abil.

146. 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.

147. Kas deklaratsioonis (39) disjunktsiooni poolte vahetamisel saadakse samaväärne deklaratsioon?

Suuremad konstruktsioonid

Selles jaotises käime läbi keerulisemad süntaktilised konstruktsioonid, mis võivad endas sisaldada mitmeid deklaratsioone.

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 operatsiooni kordumisel koodis võidakse see operatsioon ka arvutuse käigus korduvalt sooritada, kui aga selle avaldise esinemised asendada lokaalse muutujaga, millega on see avaldis lokaalse deklaratsiooniga seotud, teostatakse see arvutus 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 (valides muutujatele vastavate vaheetappide tulemust sisuliselt iseloomustavad nimed).

Põhilisim konstruktsioon lokaalsete deklaratsioonidega on *let*-avaldis kujul

let
  $\mathfrak{d}_1$
  .
  $\mathfrak{d}_l$.
in
$\mathfrak{e}$

Tema väärtuseks on avaldise $\mathfrak{e}$ väärtus eeldusel, 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
      a = x * x + 3 * x * y + 2 * y * y
      b = sin x - 3 * sin x * cos y + cos x
    in
    b / a * 100
```

on muutujaga `y` seotud korduvalt vajamineva väärtusega avaldis `x + 0.5`, et vältida sama arvutuse ja koodi kordumist. Samuti tuuakse eraldi muutujatega `a` ja `b` välja arvutatava avaldise pikemad alamavaldised.

Ülesandeid

148. 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.

149. Defineerida muutuja `täisMurdNali` väärtuseks funktsioon, mis võtab argumendiks ujukomaarvu a ja annab väärtuseks tema täisosa ja murdosa vahe. Täisarvu ujukomaarvuks teisendamiseks võib kasutada eeldefineeritud muutujat `fromIntegral`.

Kui mõnd avaldist kasutatakse korduvalt mitmes valvuris või mitmele valvurile järgnevates avaldistes, siis pole *let*-avaldisega võimalik olukorda parandada, kuna valvurikonstruktsioon ei ole avaldis. 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. *Where*-konstruktsioon on 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.

Ülesandeid

150. 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 antud arvule kooditabelis midagi ei vasta (st arv pole lõigus 0-st 255-ni), siis anda seda ütlev tekst. Ühtki avaldist ei tohi arvutada korduvalt. Kõik vajaminevad oma abimuutujad defineerida lokaalselt. Vajalikke kontrollfunktsioone otsida moodulist Data.Char.

151. Valvurikonstruktsiooni abil defineerida muutuja `täisJaMurdosa` 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.

Mis puutub arvutuste käiku, siis lokaalseid deklaratsioone iseloomustab asjaolu, et neis defineeritud näidiseid ei sobitata enne, kui mõnd sellises näidises esinevat muutujat vaja läheb. Selles mõttes käitutakse lokaalsete deklaratsioonidega nagu globaalsetegagi.

Olgu 𝔭 suvaline näidis ning 𝔞, 𝔟 suvalised avaldised. Esmapilgul paistab, et lokaalse deklaratsiooniga avaldis

```
let                                                    (40)
  𝔭 = 𝔞
in
𝔟
```

on samaväärne avaldisega

```
(\ 𝔭 -> 𝔟) 𝔞,                                          (41)
```

sest mõlema väärtus võrdub 𝔟 väärtusega eeldusel, et 𝔭 ja 𝔞 väärtused on võrdsed. Arvutuse käigus on siiski erinevus. Avaldise (40) väärtustamisel seotakse näidis 𝔭 avaldisega 𝔞, kuid ei sobitata, ja minnakse kohe avaldist 𝔟 väärtustama; näidist 𝔭 sobitama hakatakse alles siis, kui 𝔭 mõnda muutujat tarvis on. Avaldise (41) väärtustamisel aga näidis 𝔭 sobitatakse avaldise 𝔞 väärtusega enne 𝔟 juurde asumist.

See erinevus võib avalduda ka avaldise väärtuses. Kui näiteks 𝔭 = `x : xs`, 𝔞 = `[]` ja 𝔟 = `0`, siis avaldise (40) väärtus on 0, kuid (41) väärtus on $\bot$, tema väärtustamine lõpetab näidisesobituse ebaõnnestumise tõttu veateatega.

Erinevus puudub juhul, kui näidis 𝔭 on selline, mille sobitamine praktiliselt tähendabki ainult tema sidumist avaldisega: muutuja, jokker või laisk näidis.

Monaadikomprehensioon

Viimased süntaktilised konstruktsioonid avaldiste moodustamiseks, mida vaatame, kannavad komprehensioonsüntaksi (ingl *comprehension syntax*) nime. Komprehensioonsüntaksiga saab elegantse kompaktsusega väljendada küllaltki keerulisi väärtusi.

Haskellis on kaks komprehensioonsüntaksit: monaadikomprehensioon ja listikomprehensioon. Monaadikomprehensioon on üldisem, võimaldades kirja panna protseduure, liste, Maybe-tüüpi väärtusi ja üldiselt väärtusi kõikidest tüüpidest kujul $M\,A$, kus A on tüüp ja M on tüübifunktsioon, mis kuulub klassi Monad. (Protseduuride korral M = IO, Maybe-tüüpide korral M = Maybe, listide korral M = List jne.) Listikomprehensioon võimaldab kirjutada sama mis monaadikomprehensioon listide korral, kuid sisaldab ka lisavõimalusi.

Monaadikomprehensioon kannab paralleelnimetust *do*-süntaks võtmesõna **do** järgi, mis teda Haskellis alustab. Võtmesõna **do** järele kirjutatakse rida generaatoreid (ingl *generator*). Iga generaator on kas kujul `e` või kujul `p <- e`, kus `e` on avaldis tüüpi $M\,X$ ning `p` on näidis, mille muutujad on nähtavad järgmistes generaatorites, kuid mitte selles generaatoris ega eelmistes. Seejuures võib tüüp X olla igas generaatoris erinev, kuid M peab olema sama. Generaatoreid peab olema vähemalt üks ning viimane generaator peab olema paljas avaldis. Kogu *do*-avaldise tüüp langeb kokku viimases generaatoris oleva avaldise tüübiga. Lisaks generaatoritele võib *do*-süntaksi elementideks panna ka lokaalsete andmedeklaratsioonide plokke, mis algavad võtmesõnaga **let**.

Me ei hakka siin õppima monaadikomprehensioonsüntaksi tähendust üldjuhul, kuigi tähenduse ühtne kirjeldus kõigi tüübifunktsioonide M jaoks on põhimõtteliselt võimalik, vaid vaatame eraldi kolme olulisemat juhtu: protseduurid, Maybe-tüübid ja listid.

Protseduuride puhul võimaldab monaadikomprehensioon siduda mitu tegevust kokku üheks suuremaks tegevuseks. *Do*-avaldise väärtus on protseduur d, mis täidab järjest kõigi generaatorite, noole korral aga nende paremate poolte, väärtuseks olevaid protseduure. Kujul `p <- e` oleva generaatori korral sobitatakse `e` väärtuseks oleva protseduuri täitmise järel näidist `p` selle väärtusega, mille see protseduur edastab. Sobitumise korral saadakse `p` elementaarosade väärtustus, millest edaspidi saab vajadusel muutujaid arvutada, mittesobitumise korral protseduuri d täitmine katkeb täitmisaegse veaga.

Kui vigu ei teki, siis *do*-avaldise `e` väärtuseks olev protseduur edastab `e` viimase generaatori poolt edastatava väärtuse. See ei tähenda topeltedastamist, vaid niimoodi lihtsalt kujuneb selle tõttu, et generaatorid täidetakse järjekorras esimesest viimaseni.

Väärtusi, mida protseduur edastab, saab vastu võtta ainult teine protseduur. Ei ole põhimõtteliseltki võimalik kirjutada funktsioone tüübiga IO $A \to A$, mis argumendil x annaksid väärtuseks tegevuse x poolt edastatava väärtuse. Selline funktsioon rikuks ilmutatud viidatavust, sest sama abstraktne tegevus võib erinevatel kordadel edastada erinevaid väärtusi.

Mõne lihtsaid protseduure väärtuseks andva funktsiooniga oleme eelnevas juba tutvunud. Need olid muutujate `putStr`, `putStrLn` ja `print` väärtuseks. Siia võiks veel lisada muutuja

`putChar`, mille väärtuseks on funktsioon, mis võtab argumendiks sümboli ja annab protseduuri, mis kirjutab selle sümboli standardväljundisse. Nende muutujate abil võime *do*-süntaksiga kombineerida kompleksse protseduuri, mis teeb mitu sellist tegevust järjest. Näiteks avaldise

```
do
  putStrLn "Tere, Haskell!"
  putChar ' '
  putStr "2 + (-3) = "
  print (2 + (-3))
```

väärtuseks on protseduur, mis kirjutab ekraanile ühte ritta “Tere, Haskell!” ja teise ritta tühiku järele “2+(-3)=-1”.

Muutujate `getChar` ja `getLine` väärtuseks on protseduurid, mis loevad (vajadusel oodates) standardsisendist vastavalt ühe sümboli ja ühe rea ning edastavad loetud info, rea korral ilma reavahetussümbolita. Näiteks avaldise

```
do
  cs <- getLine
  c <- getChar
  putStrLn ("\nKokku " ++ cs ++ c : ".")
```
(42)

väärtuseks on protseduur, mis loeb standardsisendist kõigepealt ühe rea ja seejärel üksiku sümboli ning kirjutab siis standardväljundisse teate selle kohta, mis kokku tuli.

Siin käituvad Hugsi erinevad versioonid ja GHC erinevalt. Selleks, et saaks standardsisendist lugeda üksikut sümbolit, peab puhverdamine olema keelatud. GHC interaktiivses keskkonnas on puhverdamine vaikimisi keelatud, kuid kompilaator paneb puhverdamise vaikimisi peale. Et avaldis (42) muutujaga `main` seotuna annaks ka kompileerimisel soovitud käitumise, tuleb protseduuris esimese asjana anda käsk puhverdamine keelata.

Standardsisendi puhverdamise keelamise protseduur on väärtuseks näiteks Haskelli avaldisel

```
hSetBuffering stdin NoBuffering;
```

see tuleks avaldisse (42) esimeseks generaatoriks lisada. Kõik kolm nime, mis siin avaldises esinevad, saab moodulist System.IO. Muutuja `stdin` väärtuseks on standardsisend, konstruktor `NoBuffering` tähistab puhverdamise puudumist ning `hSetBuffering` väärtuseks on *curried*-kujul funktsioon, mis võtab argumentideks failipideme ja puhverdusviisi tähistava objekti ning annab väärtuseks protseduuri, mis seab vastava pidemega failil puhverdusviisi selliseks.

Muutuja `stdin` on tüüpi `Handle`. Konstruktori `NoBuffering` tüüp on `BufferMode` ja teised konstruktorid samast tüübist on `LineBuffering` ja `BlockBuffering`.

Kahjuks Hugsi mõned versioonid on puhverdamise koha pealt vigased ja ei lase üldse puhverdamist keelata.

Protseduuritüüpidega tuleb tegemist ka juhuarvudega opereerimisel, sest tavaliselt me soovime, et genereeritavad juhuarvud poleks programmiga üheselt määratud, erinevatel käivituskordadel peaksid tulema erinevad juhuarvud.

Juhuarvudega seonduvad operatsioonid saab moodulist System.Random. Peamiselt läheb vaja muutujaid `randomIO` ja `randomRIO`. Muutuja `randomIO` väärtus on protseduur, mis edastab ühe väärtuse, milleks võib olla võrdse tõenäosusega ükskõik milline normaalne väärtus üle kogu tüübi. Tüüp peab olema kontekstist selge või annoteeritud. Muutuja `randomRIO` väärtus on funktsioon, mis võtab argumendiks paari ja annab tulemuseks protseduuri, mis edastab ühe väärtuse, mis on võrdse tõenäosusega ükskõik milline väärtus paariga määratud lõigus. Ka siin võib olla vajalik tüüpi annoteerida. Tüüp, millest juhusuurusi genereeritakse, peab kuuluma klassi Random.

Näiteks avaldise

```
do
  x <- randomRIO (-99 , 99)
  print (x :: Int)
```

väärtus on protseduur, mis genereerib juhuslikult ühe täisarvu, mille absoluutväärtus on ülimalt kahekohaline, ja saadab selle standardväljundisse.

Ülesandeid

152. Kirjutada programm, mis küsib kasutajalt eesti keeles rida ja kui see tuleb, siis kirjutab ekraanile selle rea sümbolid vastupidises järjekorras.

153. Kirjutada programm, mis küsib kasutajalt eesti keeles sümbolit ja kohe, kui see tuleb, teatab ekraanil järgmisel real eestikeelse lausega, mis on selle sümboli kood. Lauses peavad esinema nii sümbol kui ka kood.

154. Kirjutada programm, mis väljastab ekraanile mittenegatiivse 1-st väiksema arvu täpsusega 4 kohta peale koma, mis oleks võrdse tõenäosusega ükskõik milline selline arv.

155. Kirjutada programm, mis väljastab ekraanile juhuslikult valitud ladina tähestiku tähe, mis võib võrdse tõenäosusega olla üks suurtähtedest ja üks väiketähtedest.

156. Mida teeb ja mida edastab avaldise

```
do
  putStr "A\n"
  getLine
```

väärtuseks olev protseduur?

Protseduuri poolt edastatavat väärtust saab ilmutatult seada muutuja `return` abil. Iseseisvalt on muutuja `return` väärtuseks funktsioon, mis suvalisel argumendil x annab triviaalse, tühja

protseduuri, mis lihtsalt edastab väärtuse x. Pangem tähele, et Haskelli `return` ei tähenda tagasipöördumist alamprogrammist nagu paljudes teistes keeltes. Nime `return` valik on selles mõttes ebaõnnestunud (inglise keeles *return* tähendab tagasipöördumist või tagastamist).

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                                .
      t <- getLine
      return (read s + read t :: Integer)
```
(43)

Siin kasutatud muutuja `read` väärtuseks on funktsioon, mis võtab argumendiks stringi ja annab väärtuseks täisarvu, mida see string väljendab, ehk sisuliselt parsimine. (Kuna `read` on defineeritud paljude tüüpide jaoks, täpsemalt klassi Read tüüpide jaoks, siis on annotatsioon vajalik.)

Deklaratsioon (43) 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 ++ " .")
```

Ülesandeid

157. 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. Lauses ei pea küsitud sümboleid esitama, võib rääkida “esimesest” ja “teisest” sümbolist.

158. Modifitseerida ülesandes 157 kirjutatud protseduuri edastatavat väärtust ja põhiprogrammi tööd nii, et programm sõnastaks lõpus teate kasutaja poolt sisestatud sümbolite terminites, mitte “esimese” ja “teise” terminites.

159. 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.

Vaatame edasi ülevaatlikult ka *do*-avaldist Maybe-tüüpide jaoks. Põhimõtteliselt tuleb ka siin uurida järjest generaatoreid, millise väärtuse nende avaldis annab.

Kui niisuguse *do*-avaldise mingis generaatoris oleva avaldise väärtus on Nothing, siis on kogu *do*-avaldise väärtus Nothing. Kui generaator on kujul `p <- e` ja `e` väärtus on kujul Just x, siis kui näidis `p` sobitub väärtusega x, siis uuritakse edasi järgmisi generaatoreid, kui aga näidis ei sobitu oma väärtusega, aga see väärtus on normaalne, on kogu *do*-avaldise väärtuseks Nothing. Kui generaator on paljas avaldis väärtusega kujul Just x, unustatakse ta ära. Kui kõigis generaatorites on avaldise väärtus kujul Just x ja kõik näidisesobitused õnnestuvad, siis kogu *do*-avaldise väärtus võrdub viimases generaatoris oleva avaldise väärtusega.

Muutuja `return` väärtus võrdub konstruktori `Just` väärtusega.

Olgu meil näiteks vaja kirjutada muutuja `kaheSumma` väärtuseks funktsioon, mis võtab argumendiks arvupaaride listi l ja arvud a ja b: kui listis l leiduvad paarid, mille esimesed komponendid on vastavalt a ja b, siis leiab kumbagi sorti paaridest esimese ja annab väärtuseks Just s, kus s on leitud paaride teiste komponentide summa; vastasel korral, st kui kas a või b jaoks ei leidu listis l ühtki paari, mille esimeseks komponendiks ta on, siis annab väärtuseks Nothing.

Üldplaanis sellist operatsiooni tuleb praktikas tihti vaja. Paaride list on tüüpiline andmestruktuur osalise või mitmese vastavuse kujutamiseks. Näiteks (1 , 3) : (2 , 1) : (1 , 7) : [] märgib relatsiooni, kus arvule 1 vastavad arvud 3 ja 7 ning arvule 2 vastab arv 1. Ülejäänud arvudele midagi ei vasta. Meie otsitav funktsioon peab sel listil ja argumentidel 1 ja 2 andma väärtuseks Just (3 + 1) ehk Just 4.

Paaride listist antud elemendile vastava elemendi otsimiseks on mooduli Data.List muutuja `lookup`, mille väärtus on funktsioon, mis võtab argumendiks objekti x ja paarilisti l, mille paaride esimeste komponentide tüüp langeb kokku x tüübiga: kui listis l leidub paar, mille esimene komponent on x, siis annab väärtuseks Just z, kus (x , z) on esimene paar listis l, mille vasak komponent on x; vastasel korral annab väärtuseks Nothing. Proovime algul muutujat `kaheSumma` ilma komprehensioonsüntaksita defineerida.

Näiteks kõige otsesem definitsioon oleks

```
kaheSumma l a b
  = case lookup a l of
      Just x
        -> case lookup b l of
             Just y
               -> Just (x + y)
             _
               -> Nothing
      _
        -> Nothing
```

Teine variant ilma komprehensioonsüntaksita oleks

```
kaheSumma l a b
  | isJust xo && isJust yo
    = fromJust xo + fromJust yo
  | otherwise
    = Nothing
  where
    xo = lookup a l
    yo = lookup b l
```

kus hargnemine on ühel tasemel. Muutuja `isJust` tuleb moodulist Data.Maybe ja tema väärtuseks on predikaat, mis kontrollib, kas argument on kujul Just x. Kuid ka see kood toob kaasa nii kirjutamis- kui täitmisaegseid kohmakusi.

Komprehensioonsüntaksiga saame definitsiooniks

```
kaheSumma l a b
  = do
      x <- lookup a l ,
      y <- lookup b l
      return (x + y)
```

mis on kõigi variantide seas elegantseim. Siin `x` ja `y` esinevad samas tähenduses nagu valikuavaldisega definitsioonis.

Listide korral proovitakse *do*-avaldise iga generaatoriga määratud listist läbi kõik elemendid, mis seal on. Kui generaatoris on näidis ja see antud elemendiga ei sobitu, kuid väärtus on normaalne, siis seda elementi ignoreeritakse ja uuritakse järgmisi. Tulemusena vaadatakse läbi kõik sobivate elementide kombinatsioonid; igaühe jaoks neist omandab viimane generaator väärtuseks mingi listi; *do*-avaldise väärtus on kõigi selliste listide konkatenatsioon.

Muutuja `return` väärtus on funktsioon, mis annab oma argumendil välja üheelemendilise listi, mille ainus element on see argument.

Näiteks avaldise

```
do
  x <- [1 .. 9]
  return (x * x)
```
(44)

väärtuseks on list ühekohaliste positiivsete arvude ruutudest. Tõepoolest, esimene generaator annab muutujale `x` järjest väärtused $1, 2, \ldots, 9$ ning igale sellisele väärtusele rakendatakse ruututõstmist ja tekitatakse neist üheelemendilised listid. Kogu *do*-avaldise väärtus on seega nende listide konkatenatsioon ehk $1 : 4 : \ldots : 81 : []$.

Avaldise

```
do
  x <- [1 .. 9]
  y <- [1 .. 5]
  return (x , y)
```
(45)

väärtuseks on aga list täisarvupaaridest kujul $(x\,, y)$, kus $1 \leqslant x \leqslant 9$ ja $1 \leqslant y \leqslant 5$, sorteerituna kõigepealt vasaku ja seejärel parema komponendi järgi. See järjestus näitab, et mida hilisem generaator, seda kiiremini teda vändatakse.

Kui on vaja funktsiooni, mis võtaks argumendiks listide listi ja leiaks neist kõigi mittetühjade listide pead, siis komprehensioonsüntaks on sobiv valik, võimaldades elegantselt vältida ilmutatud hargnemisi listide tühjuse-mittetühjuse järgi. Deklaratsioon

```
pead xss
  = do
      x : _ <- xss
      return x
```
(46)

defineerib just sellise funktsiooni muutuja `pead` väärtuseks. Kui argumentlisti mõni element on tühi, siis näidis `x : _` temaga ei sobitu ja ta jäetakse vahele. Näiteks `pead ["ABC", "", "hi", "!", ""]` väärtus on “Ah!”.

Kui meid huvitavad mittetühjade elementide pikkused, siis võime kirjutada deklaratsiooni

```
mittetühjadePikkused xss
  = do
      xs@ (_ : _) <- xss
      return (length xs)
```
(47)

Näiteks `mittetühjadePikkused ["ABC", "", "hi", "!", ""]` väärtus on $3 : 2 : 1 : []$.

Ülesandeid

160. Kui deklaratsioonis (46) muuta näidis `x : _` tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust?

161. Kui deklaratsioonis (47) muuta näidis `xs@ (_ : _)` tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust?

Listikomprehensioon

Kõik listide kohta kirjutatud *do*-avaldised kujul

do
$\mathfrak{g}_1$
.
.
$\mathfrak{g}_l$,

mille viimane generaator $\mathfrak{g}_l$ on kujul `return` $\mathfrak{e}$, saab samaväärselt ümber kirjutada listikomprehensioonsüntaksiga kujul

`[`$\mathfrak{e}$ `|` $\mathfrak{g}_1$`, ` $\cdots$`, ` $\mathfrak{g}_{l-1}$`]`.

Muutuja `return` argument viimasest generaatorist suundub algusse ning püstkriipsu järel tulevad teised generaatorid, mis on nüüd eraldatud komadega.

Näiteks ühekohaliste arvude ruutudest koosneva listi saab avaldisega (44) samaväärselt väljendada listikomprehensiooniga kujul

```
[x * x | x <- [1 .. 9]].
```

Listikomprehensioonsüntaks võimaldab generaatorite sekka lisada ka valvureid, mis kujutavad endast tõeväärtusetüüpi avaldisi. Valvurist mööda saab ainult juhul, kui ta väärtustub tõeseks, vastasel korral tuleb minna tagasi ja valida uus element.

Näiteks kui meid arvu 5 ruut ei huvita, siis võime selle ruutude listist välja jätta, kirjutades avaldise

```
[x * x | x <- [1 .. 9], x /= 5].
```

Avaldise

```
[(x , y) | x <- [1 .. 9], y <- [1 .. 9], x <= y]          (48)
```

väärtus on aga list arvupaaridest, mille mõlemad komponendid on ühekohalised ning teine komponent pole esimesest väiksem.

Siit on näha, miks seda süntaksit nimetatakse komprehensiooniks: ta on analoogne matemaatikast tuntud hulgakomprehensiooniga. 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.

Listikomprehensiooni kasutamisel on valvurid kasulik panna nii varakult kui võimalik, st iga valvur tuleks panna kohe nende generaatorite järele, mis määravad valvuris esinevaid muutujaid sisaldavad näidised. Kui valvuri ees on mõni generaator näidisega, millest valvur kuidagi ei sõltu, tekib tarbetu töökulu arvutuse käigus, sest valvuri väärtust kontrollitakse selle näidise kõigi väärtuste korral uuesti.

Näiteks avaldise

```
[(x , y) | x <- [1 .. 6], y <- [10 ^ x .. 100000], x >= 5]
```

väärtus on list, mille ainus element on paar $(5\,,100000)$, kuid arvutamine võtab kaua aega, sest töö käigus proovitakse läbi näiteks kõik paarid $(1\,,y)$, kus $10 \leqslant y \leqslant 100000$. Kui tuua valvur parempoolse generaatori ette, saame sama väärtusega avaldise

```
[(x , y) | x <- [1 .. 6], x >= 5, y <- [10 ^ x .. 100000]],
```

mille väärtus leitakse välkkiirelt.

Ka listikomprehensioonsüntaks lubab muude elementide vahele asetada võtmesõnaga **`let`** algavaid lokaalsete andmedeklaratsioonide plokke.

Ülesandeid

162. Kirjutada listikomprehensiooniga samaväärselt ümber avaldis (45) ning deklaratsioonid (46) ja (47).

163. Kirjutada komprehensioonsüntaksiga avaldis, mille väärtus on sama mis avaldisel (48), kuid ei sisalda valvureid.

164. Lahendada ülesanne 26 komprehensioonsüntaksi abil.

165. Kirjutada listikomprehensioonavaldis, mille väärtuseks on stringide list, mille elemendid on kahekohalised kümnendarvud suuruse järjestuses.

166. Modifitseerida ülesande 165 lahendust nii, et kõigi kahekohaliste kümnendarvude asemel on kõik kahekohalised kuueteistkümnendarvud.

167. Kasutades ära ülesandes 146 defineeritud muutujat `nihe`, defineerida muutuja `nihkesiffer` väärtuseks *curried*-kujul funktsioon, mis võtab argumendiks täisarvu n ja stringi s ning annab väärtuseks stringi, mille saab stringist s iga ladina tähe asendamisel temast tsüklilises ladina tähestikus n koha võrra tagapool oleva tähega.

168. 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.

169. 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.

170. Kirjutada avaldis, mille väärtus on kolmas täisruut, mis on suurem kui 100000. Vältida korduvaid arvutusi.

Tsükliliste arvutuste programmeerimine

Rekursioon

Rekursiivseks (ingl *recursive*) nimetatakse definitsiooni, mille parem pool sisaldab midagi, mida see definitsioon ise defineerib. Avaldise väärtustamine rekursiivse definitsiooni järgi tekitab iteratiivse protsessi: kirjutades defineeritava muutuja asemele avaldise, mille see definitsioon temaga seob, võib tulemuses esineda seesama muutuja, mida tuleb siis omakorda asendada.

Eelmises lõigus kirjeldatud rekursiivsed definitsioonid tekitavad otsese rekursiooni. Öeldakse, et kaks definitsiooni on **vastastikku rekursiivsed** (ingl *mutually recursive*), kui leidub niisugune definitsioonide jada $\mathfrak{D}_0, \mathfrak{D}_1, \ldots, \mathfrak{D}_n = \mathfrak{D}_0$, et mõlemad vaadeldavad definitsioonid esinevad selles ning iga $i = 0, \ldots, n-1$ korral $\mathfrak{D}_{i+1}$ kasutab midagi, mis on defineeritud definitsioonis $\mathfrak{D}_i$. Ka vastastikku rekursiivsed definitsioonid võivad tekitada iteratiivse protsessi analoogselt otseselt rekursiivse definitsiooniga.

Funktsionaalses paradigmas on rekursioon ainus tee tsüklilisi protsesse programmeerida. Seetõttu on rekursioon siin äärmiselt tähtis.

Efektiivsuse seisukohalt tuleb rekursiivseid definitsioone kirjutades jälgida, et rekursiivne arvutusprotsess sisaldaks ainult osi, mis sõltuvad rekursiooni parameetritest, st osi, mis erinevatel rekursioonitasemetel muutuvad. Sõltumatud osad tuleks viia rekursioonist välja, et neid sooritataks nii vähe kordi kui võimalik.

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, need moodustavad **rekursiooni baasi**.

Keerulisem-lihtsam-seos on erinevatel juhtumitel erinev, seda juba sellepärast, et argumenditüübid on erinevad. Keerulisem-lihtsam-seos võib kokku langeda arvude suurusjärjestusega — see tähendab, et funktsiooni argumendid on arvulised. Baasjuhuks on siis tüüpiliselt argument

0. Samas on võimalik **struktuurne** (ingl *structural*) rekursioon, kus funktsiooni käitumine suurematel andmestruktuuridel spetsifitseeritakse funktsiooni väärtuste kaudu väiksematel andmestruktuuridel. Andmestruktuuriks on väga tihti just listid, mispuhul definitsioon tüüpiliselt sätestab funktsiooni väärtuse mittetühjal listil funktsiooni väärtuse kaudu selle listi sabal ning tüüpilise baasjuhuna sätestatakse funktsiooni väärtus tühjal listil.

Siin tüüpilisena kirjeldatud olukorrad pole kaugeltki ainuvõimalikud ega ainumõeldavad. Arvulise argumendiga funktsioon võib olla 0-l määramata ja baasjuhuks võib olla näiteks 1. Listidel töötaval funktsioonil võib olla baasjuhuks midagi muud peale tühja listi ja rekursiivsel pöördumisel ei pruugi argumendiks anda tingimata listi saba. Keerulisem-lihtsam-seos ja baasjuhud võivad üldse puududa; sellisel juhul toimuvad rekursiivsed pöördumised küll piiramatult, kuid näiteks lõpmatu listi arvutamisel see ongi eesmärk.

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.

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
  :: (Integral a)
  => a -> a
suurSumma n
  = case compare n 0 of
      GT
        -> suurSumma (n - 1) + n ^ 4
      EQ
        -> 0
      _
        -> error "suurSumma: negatiivne liidetavate arv"
```
. (49)

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. Analoogne jutt kehtib, kui summa asemel rääkida korrutisest, ainult et argumendil 0 on sellise funktsiooni väärtus 1, kuna korrutamise ühikelement on 1.

Rekursioon võib toimuda mitme parameetri järgi korraga. Vaatame näiteks diskreetsest matemaatikast tuntud kahaneva faktoriaali funktsiooni

$$(x)_k = x \cdot (x-1) \cdot \ldots \cdot (x-(k-1)), \tag{50}$$

millel on kaks argumenti: arv x ja naturaalarv k. Kui $k > 0$, siis saab seose (50) paremas pooles eraldada esimese teguri x ning 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

```
kahFact
  :: (Num a, Integral b)
  => a -> b -> a
kahFact x k
  = case compare k 0 of
      GT
        -> x * kahFact (x - 1) (k - 1)
      EQ
        -> 1
      _
        -> error "kahFact: negatiivne teine argument"
```
. (51)

Rekursiivne definitsioon tuleneb tihti otseselt matemaatikas kasutatavast rekurrentsest võrrandist. Kuigi Haskell on väljendusvahendina küllalt võimas, et tõlkida sellesse matemaatikas antavaid definitsioone, pole matemaatikas standardse definitsiooni otsene ülevõtmine alati arvutuslikult rahuldav. Matemaatikas on hea definitsiooni kriteeriumiks lihtne sisuline mõistetavus, võibolla ka lühidus, kuid arvutuslikul aspektil seal kaalu pole. Programmeerimisel tuleb rekursiooniskeemi valikul ka arvutuskeerukust silmas pidada.

Võtame näiteks Fibonacci jada, mis matemaatiliselt antakse seostega

$$F_0 = 0, \qquad F_1 = 1, \qquad \forall n \geqslant 2\,(F_n = F_{n-1} + F_{n-2}).$$

See jada üldistub loomuldasa ka negatiivsetele indeksitele valemiga

$$\forall n < 0\,\left(F_n = (-1)^{n+1} F_{-n}\right).$$

Kui tõlgiksime need matemaatilised seosed otse Haskelli, võiksime saada näiteks koodi

```
fib
  :: (Integral a)
  => a -> a
fib n
  | n >= 2
    = fib (n - 1) + fib (n - 2)
  | n >= 0
    = n
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
. (52)

Kasutatud muutuja `odd` moodulist Prelude väärtus on predikaat, mis kontrollib, kas argument on paaritu arv.

Selle koodi järgi arvutades toob iga `fib` väljakutse 1-st suurema väärtusega argumendil kaasa kaks uut väljakutset; kuna need väljakutsed töödeldakse teineteisest sõltumatult, kasvab väljakutsete arv eksponentsiaalselt.

Lineaarse keerukuse saavutamiseks tuleks pidevalt hoida kaks viimast vahetulemust korraga kättesaadavana, siis piisaks igal tasemel ühest rekursiivsest väljakutsest. Otseseim võimalus selleks on muuta definitsiooni selliselt, et defineeritav funktsioon annaks välja paare kahest järjestikusest Fibonacci arvust. Õige funktsiooni väärtus avalduks siis kui selle funktsiooni väärtuse üks komponent.

Paaride sissetoomisel saadav funktsioon on põhimõtteliselt abifunktsioon, mistõttu defineerime ta lokaalse deklaratsiooniga õige funktsiooni definitsiooni kehas. Definitsiooniks saame

```
fib n
  | n >= 0
    = let
        fib 0
          = (0 , 1)
        fib n
          = let
              (u , v)
                = fib (n - 1)
            in
            (v , u + v)
      in
      fst (fib n)
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```
. (53)

Siin on kaks *let*-avaldist üksteise sees. Neist välimine defineerib lokaalse muutuja `fib` väärtuseks abifunktsiooni, mille väärtused on paarid kahest järjestikusest Fibonacci arvust. Sisemise *let*-avaldise abil esitatakse suvaline paar kahest järjestikusest Fibonacci arvust eelmise sellise paari kaudu.

Definitsioon (53) näitab muuhulgas seda, et lokaalselt saab varjata isegi sellessamas deklaratsioonis defineeritava muutuja. See definitsioon defineerib globaalset muutujat `fib`, kuid esimesele valvurile vastav parem pool defineerib lokaalselt samanimelise muutuja, mille väärtus on hoopis teine.

Lokaalse muutuja nimi võiks olla ka midagi muud, kuid antud juhul polnud tarvidust uut nime välja mõelda. Globaalses tähenduses on `fib` ainult kahes kohas: kõige alguses ja lõpu eel rakendatuna argumendile `-n`. Kõik teised kasutused on lokaalses tähenduses.

Ülesandeid

171. Arvu n faktoriaal $n!$ avaldub kahaneva faktoriaali kaudu seosega $n! = (n)_n$. Seetõttu saab definitsiooniga (51) antud muutujat `kahFact` kasutades kodeerida faktoriaalifunktsiooni definitsiooniga

```
fact n
  = kahFact n n
```

Anda muutujale `fact` samaväärne rekursiivne definitsioon ilma muutujat `kahFact` kasutamata.

172. Defineerida muutuja `fibKorrutis` väärtuseks funktsioon, mis võtab argumendiks täisarvu n: kui n on mittenegatiivne, siis annab väärtuseks Fibonacci arvude F_1 kuni F_n korrutise; negatiivse n korral lõpetab arvutus töö olukorda iseloomustava veateatega.

173. Kirjutada muutujale `kahFact` koodiga (51) samaväärne definitsioon, mis põhineks rekursiooniskeemil, mis igal sammul eraldab viimase teguri.

174. Arvu x **kasvavaks faktoriaaliks** k järgi nimetatakse arvu $(x)^k = x{\cdot}(x{+}1){\cdot}\ldots{\cdot}(x{+}(k{-}1))$. Defineerida muutuja `kasvFact` väärtuseks kasvava faktoriaali funktsioon *curried*-kujul.

175. Defineerida muutuja `maxMod` väärtuseks *curried*-kujuline 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.

176. **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 a annab väärtuseks L_a. Arvutus peab toimuma lineaarse keerukusega.

Kõik funktsioonid listidel, mis teevad midagi määramata arvu komponentidega, 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.

Olgu meil näiteks 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 kõigepealt originaallisti nullidest omaette list ja loetakse siis sealt need nullid kokku.

Rekursiooniga on võimalik defineerida efektiivsem arvutus, mis ei tekitaks mõttetut nullide vahelisti, vaid loendaks neid originaallisti peal. Defineerime funktsiooni, mis võtab argumendiks suvalise arvulisti ja annab tulemuseks tema nullide arvu, muutuja `nullideArv` väärtuseks. 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 koodi

```
nullideArv
  :: (Num a)
  => [a] -> Int
nullideArv (x : xs)
  | x == 0
    = 1 + ülejäänu
  | otherwise          .                    (54)
    = ülejäänu
  where
    ülejäänu
      = nullideArv xs
nullideArv _
  = 0
```

Võtame veel näiteks avaldised kujul

```
(elem α [ , delete α []).                   (55)
```

Moodulist Prelude tuleva muutuja `elem` väärtuseks on funktsioon, mis võtab argumendiks objekti x ja sama tüüpi elementidega listi l: kui objekt x esineb listis l, annab ta väärtuseks True, vastasel korral kui l on lõplik, siis annab väärtuseks False. Moodulist Data.List tuleva muutuja `delete` väärtus on funktsioon, mis võtab samasugused argumendid x ja l: kui objekt x esineb listis l, siis annab väärtuseks listi, mille saab listist l objekti x esimese esinemise väljajätmisel, vastasel korral annab väärtuseks l.

Nende funktsioonide arvutamisel tehtav töö on suures osas ühine, sest otsitakse sama elementi samast listist. Seega pole avaldise (55) väärtustamine optimaalne tee tema väärtuseks oleva paari leidmiseks. Kui sellist paari on vaja, oleks mõttekas arvutada soovitud tõeväärtus ja list ühekorraga, mitte teineteisest sõltumatult.

Defineerimegi nüüd muutuja `eemaldaEsimene` väärtuseks sellise funktsiooni, mis võtab samasugused argumendid x ja l nagu `elem` ja `delete` väärtuseks olevad funktsioonid ja annab tulemuseks paari, mille esimene komponent võrdub neist kahest funktsioonist esimese väärtusega argumentidel x ja l ning teine komponent teise funktsiooni väärtusega samadel argumentidel,

kusjuures paari arvutamisel toimub elemendi otsimine listist ainult ühe korra. Kood võiks olla

```
eemaldaEsimene
  :: (Eq a)
  => a -> [a] -> (Bool , [a])
eemaldaEsimene a (x : xs)
  | a == x
    = (True , xs)
  | otherwise
    = let
        (tv , us)
          = eemaldaEsimene a xs
      in
      (tv , x : us)
eemaldaEsimene _ _
  = (False , [])
```
. (56)

Nüüd võime avaldise `(elem a l , delete a l)` asemel väärtustada avaldise `eemaldaEsimene a l`.

Definitsiooni (56) saab veel pisut parandada. Paneme tähele, et defineeritava muutuja esimene argument antakse uuel väljakutsel alati muutmata kujul edasi: ainus rekursiivne väljakutse asub *let*-plokis ja argument seal on `a`, mis langeb kokku argumendiga deklaratsiooni päises. Järelikult ei peaks seda argumenti rekursioonis üldse olema. Viies rekursiooni lokaalsesse deklaratsiooni, saame definitsiooni

```
eemaldaEsimene a xs
  = let
      eemaldaEsimene (x : xs)
        | a == x
          = (True , xs)
        | otherwise
          = let
              (tv , us)
                = eemaldaEsimene xs
            in
            (tv , x : us)
      eemaldaEsimene _
        = (False , [])
    in
    eemaldaEsimene xs
```
, (57)

milles rekursioon muutumatu väärtusega argumenti kaasas ei kanna. Selline optimisatsioon parandab arvutuskiirust siiski vaid marginaalselt.

Ülesandeid

177. Otsida Hugsi teegist muutujate `!!`, `take`, `drop`, `splitAt` definitsioonid ja saada neist aru.

178. Defineerida muutuja `suurtähega` väärtuseks funktsioon, mis võtab argumendiks stringide listi ja annab tulemuseks suurtähega algavate stringide arvu selles.

179. Defineerida muutuja `esitähed` väärtuseks funktsioon, mis võtab argumendiks stringide listi ja annab tulemuseks nende stringide algusosadest moodustatud lühendi. Stringidest, kus on vähemalt kaks tähte, peab lühendisse tulema parajasti kaks esimest tähte; ühetähelistest stringidest peab tulema nende ainus täht; tühjad stringid annavad lühendisse tühiku.

180. Defineerida muutuja `eemaldaKõik` väärtuseks *curried*-kujul funktsioon, mis võtab argumendiks objekti x ja sama tüüpi objektide listi l ning annab väärtuseks paari, mille esimene komponent on tõeväärtus, mis ütleb, kas x esineb listis l, ja teine komponent on list, mis saadakse l-st kõigi x esinemiste eemaldamisel. Minimiseerida korduvat tööd arvutuse käigus.

181. Defineerida muutuja `eemaldaJaLoenda` väärtuseks *curried*-kujul funktsioon, mis võtab samasugused argumendid x ja l nagu ülesande 180 funktsioon ning annab väärtuseks paari, mille esimene komponent on objekti x esinemiste arv listis l ja teine komponent on list, mis saadakse l-st kõigi x esinemiste eemaldamisel. Minimiseerida korduvat tööd arvutuse käigus.

182. Defineerida muutuja `kaheksPiiriJärgi` väärtuseks *curried*-kujuline funktsioon, mis võtab argumendiks objekti n mingist järjestusega tüübist ja sama tüüpi objektide listi l ning annab tulemuseks listipaari, kus esimeses komponendis on listi l kõik need elemendid, mis on n-st väiksemad, teises komponendis ülejäänud.

183. Defineerida muutuja `korrutisteSumma` väärtuseks funktsioon, mis võtab argumendiks arvulisti ja annab tulemuseks tema elementide paarikaupa summade korrutise.

184. Defineerida muutuja `võrdlePikkust` väärtuseks *curried*-kujuline 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.

Mõnikord ei piisa listirekursiooni puhul sellest, et programmeeritakse rekursiooni baas vaid tühja listi jaoks. Olgu meil näiteks vaja defineerida muutuja `summad` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja leiab selle listi kahe järjestikuse elemendi summade listi. Et ühtki summat leida, peab olema vähemalt kaks elementi, seetõttu on nii tühja kui üheelemendilise listi

juhud vaja käsitleda üldjuhust eraldi. Sobib definitsioon

```
summad
  :: (Num a)
  => [a] -> [a]
summad (x : xs@ (y : _)).                                    (58)
  = x + y : summad xs
summad _
  = []
```

Kuna funktsiooni väärtus tühjal ja üheelemendilisel listil on võrdne tühja listiga (ühtki summat pole), õnnestub need kaks juhtu kokku võtta ja piirduda kahe deklaratsiooniga.

Toomaks näidet veel pisut ebastandardsemast rekursiooniskeemist, defineerime muutuja `segmendid` väärtuseks funktsiooni, mis argumendiks saadud listi järgi koostab tema segmentide listi, kus listi segmendiks nimetame maksimaalseid mittekahanevaid lõike listis. 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                                                (59)
    = let
        (us , vs)
          = eraldaSegment xs
      in
      (x : us , vs)
eraldaSegment _
  = ([] , [])
```

annab muutuja `eraldaSegment` väärtuseks funktsiooni, mis võtab argumendiks listi ja lõhub selle kaheks listiks kohalt, kus lõpeb esimene segment. 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                          .                    (60)
        (us , vs)
          = eraldaSegment xs
      in
      us : segmendid vs
```

Definitsioon (60) on küll rekursiivne, sest teise valvuri juht sisaldab pöördumist `segments` poole, kuid rekursiooniskeem on ebatüüpiline, kuna rekursiivsel pöördumisel ei anta argumendiks listi saba. Kui argument on mittetühi list — käiku läheb siis teine deklaratsioon, sest esimese deklaratsiooni argumendinäidis temaga ei sobitu —, kutsutakse välja funktsioon `eraldaSegment`, mis annab kätte esimese segmendi ning järelejääva osa, ning rekursiivsel pöördumisel antakse argumendiks viimane.

Ülesandeid

185. Defineerida muutuja `järjestatud` väärtuseks funktsioon, mis võtab argumendiks arvude listi ja annab väärtuseks tõeväärtuse vastavalt sellele, kas list on lõplik ja mittekahanevalt järjestatud või leidub listis element, mis on talle eelnevast väiksem.

186. 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.

187. Defineerida muutuja `summad'` väärtuseks funktsioon, mis on nagu koodiga (58) defineeritud muutuja `summad` väärtus, kuid mittetühja argumentlisti puhul on tulemuslistis üks element rohkem ja see võrdub argumentlisti viimase elemendiga.

188. 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.

189. 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.

190. Defineerida muutuja `needi` väärtuseks *curried*-kujul 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.

191. Kasutades ülesandes 180 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.

192. Kas definitsioonis (60) esimese valvuri juhu ärajätmisel saame samaväärse definitsiooni?

193. Kas definitsioonis (60) esimese valvuri juhus `[]` asendamisel `xs`-ga saame samaväärse definitsiooni?

Olgu meil nüüd vaja muutuja `kaheksLoe` väärtuseks defineerida funktsioon, mis võtab argumendiks listi l ja annab tulemuseks listipaari, kus l elemendid on üle ühe jaotatud kahte listi. Sellise jaotamise tegemisel tuleb aga igal sammul teada, kummasse listi käsilolev element paigutada. Probleemiks on, kuidas seda infot hoida.

Ühe lahenduse pakub vastastikrekursioon. Kirjutame kaks definitsiooni, millest üks on elemendi paigutamiseks vasakpoolsesse, teine aga parempoolsesse listi. Saame koodi

```
kaheksLoe
  :: [a] -> ([a] , [a])
kaheksLoe (x : xs)
  = let
      (us , vs)
        = kaheksLoe' xs
    in
    (x : us , vs)
kaheksLoe _
  = ([] , [])
```

.

(61)

```
kaheksLoe' (x : xs)
  = let
      (us , vs)
        = kaheksLoe xs
    in
    (us , x : vs)
kaheksLoe' _
  = ([] , [])
```

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. Haskellis võivad nimelt vastastikku rekursiivsed olla ka definitsioon ja tema kehas esinev lokaalne definitsioon.

Antud ülesande lahendamisel leidub vastastikrekursioonile aga ka elegantne alternatiiv: igal sammul vahetada ehitatava paari komponendid. Selle idee realiseerib kood

```
kaheksLoe (x : xs)
  = let
      (us , vs)
        = kaheksLoe xs
    in
    (x : vs , us)
kaheksLoe _
  = ([] , [])
```
. (62)

Niisugune lahendus on võimalik tänu sellele, et rekursiooni baasjuht on kahe variandi suhtes sümmeetriline: baasjuhul me ei pea teadma, kumb komponent saab lõpuks olema vasak ja kumb parem.

Ülesandeid

194. Defineerida muutuja `vahelduv` väärtuseks funktsioon, mis võtab argumendiks stringi ja annab väärtuseks stringi, mille igal paarituarvulisel kohal on algse stringi vastaval kohal oleva tähe suurtäheline variant ning igal paarisarvulisel kohal algse stringi vastaval kohal oleva tähe väiketäheline variant.

195. Nii vastastikrekursiooni abil kui ilma defineerida muutuja `vaheliti` väärtuseks *curried*-kujul 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.

196. Defineerida muutuja `otstestKeskele` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks listi, kus argumentlisti elemendid on paigutatud ringi selliselt, et esimene element on jäänud esimeseks, viimane on läinud teiseks, teine kolmandaks, tagant teine neljandaks jne.

Rekursiivselt defineeritud protseduurid

Nagu juba öeldud, pole Haskellis tsükliliste arvutuste programmeerimiseks muud valikut kui kasutada rekursiooni. Muuhulgas kehtib see tsüklis jooksva interaktiivse protsessi kohta.

Funktsiooni rekursiivse defineerimisega oleme juba tutvunud, tsüklis jooksvat suhtlust keskkonnaga on võimalik niimoodi programmeerida. Olgu meil näiteks vaja protseduuri, mis kuulab standardsisendist sümboleid ja lõpetab niipea, kui sisestatakse teatav kindel sümbol. Selleks võib kodeerida funktsiooni, mis võtab argumendiks selle sümboli, mille peale töö lõppema peab, ja annab tulemuseks vastava protseduuri. Sobib kood

```
kuulaKuni
  :: Char -> IO ()
kuulaKuni a
  = do
      c <- getChar
      if c == a then putChar '\n' else kuulaKuni a
```
. (63)

Kui standardsisendist loetakse oodatav sümbol, siis kirjutab protseduur standardväljundisse reavahetuse, et ekraanipilt ilusam jääks, ja lõpetab, vastasel korral jätkab algusest.

Seda protseduuri kasutava programmi näiteks võiks olla

```
main
  = do
      hSetBuffering stdin NoBuffering
      kuulaKuni 'X'
```
.

See programm ootab standardsisendist sümboleid kuni suure X-täheni, siis lõpetab.

Paneme tähele, et koodis (63) toimub rekursiivne pöördumine muutuja `kuulaKuni` poole alati sama argumendiga. See tähendab, et siin puudub igasugune argumendi lihtsustumine rekursiivse arvutuse käigus, mida varem alati soovisime. Arvutusprotsessi lõpetamine sõltub siin peale argumendi ka muudest asjaoludest.

Kuna argument ei muutu, ei pruugi teda ka pidevalt kaasas kanda. Teeme seetõttu siin sama optimeeringu, millega muutuja `eemaldaEsimene` definitsioonist (56) sai definitsioon (57): viime rekursiooni lokaalsesse definitsiooni, jättes argumendi `a` maha. Tulemuseks on kood

```
kuulaKuni a
  = let
      proc
        = do
            c <- getChar
            if c == a then putChar '\n' else proc
    in
    proc
```
,

milles defineeritakse rekursiivselt lokaalne muutuja `proc`, mille väärtus on mitte funktsioon, vaid hoopis protseduur.

Natuke keerulisem, kuid samal moel lahendatav on olukord, kui protseduur peab töö käigus konstrueerima mingit väärtust ja selle töö lõpus edastama. Olgu meil näiteks vaja protseduuri, mis kuulab standardsisendit, kuni sisestatakse tühi rida, leiab sisestatud ridade arvu ning edastab selle väärtuse. Selleks võime kirjutada koodi

```
loendaRead
  :: (Integral a)
  => IO a
loendaRead
  = do
      rida <- getLine                    ,                    (64)
      if null rida
         then return 0
         else do
                ridadeArv <- loendaRead
                return (1 + ridadeArv)
```

kus jällegi on rekursiivne protseduuridefinitsioon.

Ülesandeid

197. 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.

    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. Jälgida, et ekraanipilt oleks loetav.

198. Defineerida muutuja `koodid1` väärtuseks protseduur, mis ootab standardsisendist sümboleid ja iga sümboli järele kirjutab ekraanile tema koodi. Iga sümboli info peab tulema eraldi reale. Töö lõpeb, kui sisestatakse sümbol, mis on töö käigus juba analüüsitud.

199. Defineerida muutuja `koodid2` väärtuseks protseduur, mis töötab sarnaselt ülesande 198 lahendusega, kuid lõpetab töö vaid juhul, kui sama sümbolit sisestatakse kaks korda järjest.

200. Defineerida muutuja `pööra` väärtuseks protseduur, mis tsüklis küsib loomulikus keeles kasutajalt sisendit, loeb klaviatuurilt rea ja toob rea lõppedes ekraanile selle rea sümbolid vastupidises järjekorras, kuni sisestatakse tühi rida.

201. Kirjutada programm, millega saab testida koodiga (64) defineeritud muutujat `loendaRead`.

202. Defineerida muutuja `loendaSümbolid` väärtuseks protseduur, mis kuulab standardsisendit kuni tühja reani ja edastab nendes olevate sümbolite koguarvu. Kirjutada programm, millega seda muutujat testida.

203. Kasutades ülesande 150 lahendust, defineerida muutuja `diagnoosiSümboleid` väärtuseks protseduur, mis tsüklis ootab klaviatuurilt sümboleid ja igaühe kohta ükshaaval teadustab ekraanil, kas tegu on väiketähega, suurtähega, numbriga või millegi muuga. Protseduuri töö lõpeb, kui sisestatakse *enter*. Ekraanipilt peab olema loetav.

204. Defineerida muutuja `mäng` väärtuseks *curried*-kujuline funktsioon, mis võtab argumendiks kaks protseduuri, mis mõlemad edastavad tõeväärtuse, ja annab tulemuseks protseduuri, mis paneb need protseduurid omavahel järgnevalt kirjeldatavat mängu mängima ja edastab võitja.

    Mäng algab sellega, et mõlemad protseduurid täidetakse ühe korra. Kui nende edastatud väärtused on erinevad, siis on tõese edastanud protseduur kohe võitnud. Vastasel korral mängitakse sama mängu algusest peale uuesti.

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 on paratamatult iteratiivne ja mida funktsionaalses keeles järelikult ilma rekursioonita väljendada ei ole võimalik.

Proovime kirjeldada funktsiooni, mis genereerib geomeetrilise jada: defineerime muutuja `geom`, mille väärtuseks oleks *curried*-kujul funktsioon, mis võtab argumendiks arvud a ja q ning annab tulemuseks lõpmatu listi, milles on järjest kõik liikmed geomeetrilisest jadast algelemendiga a teguriga q. Seda on võimalik teha mitme ideega, aga siin võtame aluseks tähelepaneku, et konstrueeritava jada iga liige alates teisest on eelmise q-kordne. Saame koodi

```
geom
  :: (Num a)
  => a -> a -> [a]                .                    (65)
geom a q
  = a : [x * q | x <- geom a q]
```

Arvutus selle definitsiooni järgi toimub järgmiselt. Saades argumendid, pannakse kõigepealt paika listi pea. Listi saba arvutamiseks hakatakse uuesti arvutama avaldist `geom a q`. Kuna argumendid on samad, kopeerib see alamarvutus kogu arvutust. Seega alamarvutus suudab samuti

leida esimese elemendi. Kuid sellest on küll, et põhiarvutus saaks definitsioonist (65) määrata listi teise elemendi. Kuna alamarvutus kopeerib põhiarvutust, suudab teise elemendi leida ka alamarvutus. Kuid teise elemendi põhjal leiab põhiarvutus kolmanda elemendi. Nii jätkates leitakse listi elemendid kuitahes kaugele.

Selles väärtustusprotsessis pole midagi põhimõtteliselt uut, kõik toimub varemkäsitletud reeglite alusel.

Kuna argumendid rekursiivsete pöördumiste käigus jäävad samaks, võime nüüdki teha samalaadse optimeeringu nagu definitsioonide (56) ja (63) puhul: viime rekursiooni lokaalsesse definitsiooni, jättes argumendid `a` ja `q` maha. Tulemuseks on kood

```
geom a q
  = let
      gs
        = a : [x * q | x <- gs]
    in
    gs
```
(66)

Lokaalne deklaratsioon defineerib rekursiivselt muutuja `gs`, mille väärtus on list.

Oluline on näha, et siin pole tegemist enam marginaalse võiduga nagu eelmistel analoogsetel kordadel, vaid keerukus alaneb oluliselt. Selles võib veenduda interaktiivses keskkonnas katsetades. Põhjuseni pole keeruline jõuda. Arvutades definitsiooni (65) järgi, käivitatakse listi saba arvutamiseks sõltumatu kopeeriv arvutus, mis alates kolmanda elemendi arvutamisest järelikult käivitab omakorda kopeeriva arvutuse jne. Iga elemendi leidmiseks hakatakse kogu listi otsast peale arvutama, mistõttu arvutus on ruutkeerukusega. Arvutades aga definitsiooni (66) järgi, mingeid koopiaid ei teki ja toimub 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 on võimalik, sest “kirjutav” mehhanism on “lugevast” ühe lüli võrra ees. Seetõttu arvutatakse ainult üks list ja arvutus on lineaarse keerukusega.

See, et rekursiivsete väljakutsete käigus ei toimu argumentide lihtsustumist, ei ole meie jaoks enam uus nähtus, seda nägime juba protseduuride rekursiivsel defineerimisel. Kuid definitsioonidel (65) ja (66) on veel üks omapära: ehkki rekursiivsed, ei sisalda nad hargnemiskonstruktsioone. Seni on hargnemine rekursiooniskeemis ehk paistnud koodi mõistlikkuse eeldusena, hargnemine on olnud vajalik selleks, et baasjuhtudel rekursioon lõpetada. Definitsioonidel (65) ja (66) aga ei näi baasjuhtu üldse olevat, rekursiivne pöördumine toimub iga argumendi puhul, paratamatult. Selline definitsioon on mõttekas tänu sellele, et arvutus annab pidevalt välja uusi komponente lõpmatust listist.

Listi rekursiivsel määratlemisel omandavad rekursiooni samm ja rekursiooni baas varasemast erineva tähenduse. Definitsiooni kirjutades tuleb mõelda, mis on listi pea ja kuidas listi saba avaldub terve listi kaudu — või, keerulisemal juhul, mis on listi mingi mittetühi algusjupp ja kuidas listi ülejääv osa avaldub kogu listi kaudu. Algusosa määratlus mängib rekursiooni baasi

rolli ning ülejääva osa avaldamine kogu listi kaudu rekursiooni sammu rolli. Nii võttes on ka siin baasi olemasolu vajalik, sest ilma selleta lististruktuuri ei teki.

Lihtsaimad rekursiivsed definitsioonid, mis määratlevad mingi muutuja väärtuseks listi, on sellised, kus see list on konstantne. Näiteks lõpmatul listil, mille iga element on 0, võrdub pea 0-ga ja saba listi endaga. Siit saame deklaratsiooni

```
zs
  = 0 : zs.
```
(67)

Vaatame ka keerulisemaid näiteid. Oletame, et tahame arvutada listi, mille komponentideks 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.

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** (ingl *merge*). Selle saame koodiga

```
põimi xs@ (a : as) ys@ (b : bs)
  = case compare a b of
      LT
        -> a : põimi as ys
      GT
        -> b : põimi xs bs
      _
        -> a : põimi as bs
põimi xs           []
  = xs
põimi _            ys
  = ys
```
(68)

Nüüd võime otsitava listi muutuja `kords` väärtuseks anda definitsiooniga

```
kords
  = 1 : põimi [x * 2 | x <- kords] [x * 5 | x <- kords].
```

Defineerime sama tehnikaga muutuja `fibs` väärtuseks listi, mis sisaldab parajasti kogu Fibonacci jada. Sobib definitsioon

```
fibs
  = 0 : 1 : summad fibs'
```
(69)

kus muutuja `summad` on antud definitsiooniga (58).

Definitsiooni (69) järgi toimub Fibonacci arvude arvutamine lineaarse ajaga järjekorranumbri suhtes, iga järgneva elemendi leidmiseks tehakse vaid üks liitmine. Definitsiooni (53) kõrvale võib seega tuua veel ühe n-ndat Fibonacci arvu lineaarse keerukusega arvutamist realiseeriva definitsiooni

```
fib n
  | n >= 0
    = fibs !! n                              .
  | otherwise
    = (if odd n then 1 else -1) * fib (-n)
```

Ülesandeid

205. Otsida Hugsi teegist üles muutujate `repeat` ja `cycle` definitsioonid ja saada neist aru.

206. Ülesandes 176 defineeriti Lucas' jada. Defineerida muutuja `lucs` väärtuseks list, mille elementideks on järjest kõik Lucas' jada elemendid.

207. Defineerida muutuja `hamming` väärtuseks list, mille elementideks on kasvavas järjekorras kõik positiivsed täisarvud, mille kanoonilises esituses ei esine muud algarvud peale 2, 3, 5.

208. Defineerida muutuja `tuia` väärtuseks lõpmatu list elementidega $1, 2, 3, 2, 3, 4, 3, 4, 5, \ldots$.

209. 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.

210. Defineerida muutuja `bitistringid` väärtuseks list, mille elementideks on kõik lõplikud bitistringid, igaüks üks kord.

211. Defineerida muutuja `rotatsioonid` väärtuseks funktsioon, mis võtab argumendiks listi ja kui see on lõplik, siis annab tulemuseks tema kõik tsüklilised nihked listina, mille pikkus võrdub argumentlisti pikkusega.

Et rekursiivselt defineeritud andmestruktuur ei pea tingimata lõpmatu olema, võime veenduda Collatzi jadade näitel. 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 rohkem liikmeid polegi;
- 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.

Definitsioonist üldse ei selgugi, kas Collatzi jada on lõplik või lõpmatu. Samas on lihtne tõlkida Collatzi jada arvutusreeglid Haskelli deklaratsiooniks, mis rekursiivselt määratleb listid, mille elementideks on Collatzi jada liikmed. Koodiks sobib

```
collatz
  :: (Integral a)
  => a -> [a]
collatz n
  | n > 0
    = let
        f (x : xs)
          | x > 1
            = (if even x then x `div` 2 else x * 3 + 1) : f xs .
          | otherwise
            = []
        cs
          = n : f cs
      in
      cs
  | otherwise
    = error "collatz: mittepositiivne argument"
```

(70)

Lokaalset muutujat `f` kutsutakse välja ainult mittetühjadel listidel, mistõttu pole vajadust tühja argumentlisti juhu lisamisele tema definitsiooni.

Interaktiivses keskkonnas võib nüüd kergesti erinevaid Collatzi jadu tervikuna välja arvutada ja näha, et nad kipuvad lõppema. Matemaatikas oli küsimus, kas leidub selline n, mille korral Collatzi jada on lõpmatu, kaua vastuseta. Lõpuks õnnestus siiski tõestada, et Collatzi jada on lõplik iga n korral.

Viimase näitena olgu meil vaja defineerida muutuja `üleÜheEtte` väärtuseks funktsioon, mis võtab argumendiks listi l ja annab tulemuseks listi, mille alguses on listi l elemendid üle ühe alates peast ja seejärel l ülejäänud elemendid.

Esimene võimalus seda teha on kasutada definitsiooniga (61) või (62) antud muutujat `kaheksLoe`. Tema rakendamine argumentlistile teeb jaotamistöö ära ja jääb üle vaid paari

komponendid ühte listi kokku konkateneerida. Seda väljendab kood

```
üleÜheEtte
  :: [a] -> [a]
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 esimese listi mitte suvalisse kohta, vaid teise listi ette. Selleks tuleb `kaheksLoe` definitsiooni baasjuhtudes panna paari esimeseks komponendiks tulemuspaari teine komponent, mitte tühi list nagu varem. Võttes aluseks vastastikrekursiivse definitsiooni (61), saame `üleÜheEtte` uueks definitsiooniks

```
üleÜheEtte xs
  = let
      kaheksLoe0 (x : xs)
        = let
            (us , vs)
              = kaheksLoe1 xs
          in
          (x : us , vs)
      kaheksLoe0 _
        = (vs , [])
      kaheksLoe1 (x : xs)
        = let
            (us , vs)
              = kaheksLoe0 xs
          in
          (us , x : vs)
      kaheksLoe1 _
        = (vs , [])
      (us , vs)
        = kaheksLoe0 xs
    in
    us
```

(71)

Siin on lokaalsete muutujate `kaheksLoe0`, `kaheksLoe1` ja `vs` definitsioonid omavahel vastastikrekursiivsed. Neist esimese kahe muutuja väärtus on funktsioon, kolmandal aga list. Seejuures käib `vs` defineerimine läbi paari, mida seetõttu ka võib lugeda rekursiivselt defineerituks.

Ülesandeid

212. Näidata definitsioonis (71) nimede `us` ja `vs` kõigi esinemiste kohta nende tähendusulatus (skoop).

213. Kirjutada definitsioon (71) samaväärselt ümber ilma paarinäidisteta, kasutades muutujaid `fst` ja `snd`. Millised muutujad on nüüd defineeritud vastastikrekursiivselt ja mis on nende väärtus?

214. Defineerida muutuja `paarisEtte` väärtuseks funktsioon, mis võtab argumendiks täisarvude listi l ja annab tulemuseks listi, mille alguses on l paaris elemendid ja seejärel l paaritud elemendid. Vältida tühja tööd arvutusel.

215. Defineerida muutuja `kolmeJärgi` väärtuseks funktsioon, mis võtab argumendiks täisarvude listi l ja annab tulemuseks listi, mille alguses on l elemendid, mis annavad 3-ga jagades jäägi 0, nende järel elemendid, mis annavad jäägi 1, ning lõpus elemendid, mis annavad jäägi 2. Vältida tühja tööd arvutusel.

216. Defineerida muutuja, mille väärtustamisel tekib lõpmata palju veateateid.

Arvutamine rekursiivse funktsiooni argumentidel

Imperatiivses keeles programmeerides hoitakse arvutuse jaoks pidevalt vajaminevad andmed tüüpiliselt muutujates ja vajadusel uuendatakse neid andmeid omistuskäskudega. Funktsionaalses keeles pole muutujaid sellises mõttes nagu on imperatiivses keeles, omistus puudub. Kuid sellegipoolest on funktsionaalseski keeles võimalik arvutamine samasugusel põhimõttel, kasutades jooksvate suuruste salvestamiseks rekursiivse funktsiooni argumente.

Taolised rekursiivse funktsiooni argumendid vastanduvad senivaadeldutele selles mõttes, et nad rekursiivsete väljakutsete käigus ei lihtsustu, tüüpiliselt on hoopis vastupidi. Nende roll pole näidata, millal rekursiivsete pöördumiste jada lõpetada, see otsus tehakse teiste argumentide või mõne spetsiaalse lõpetamistingimuse põhjal. Tihti on need argumendid spetsiaalselt jooksvate andmete hoidmiseks kunstlikult lisatud, mitte ei tulene ülesande püstitusest.

Järjehoidjad

Lihtsamal juhul on salvestava argumendi ülesandeks tööjärje meelespidamine. Nimetame sellist parameetrit järjehoidjaks.

Pöördume tagasi muutuja `kaheksLoe` defineerimise juurde. Esimene vaadeldud definitsioon (61) põhines vastastikrekursioonil, mis nõudis korraga kahe funktsiooni kodeerimist. Teine definitsioon (62) nõudis poole vähem tööd, kuid eeldas suhteliselt nutikat programmeerijat.

Et korraga vähendada töömahtu ja vajaminevat kavalust, töötame ümber vastastikrekursiooniga definitsiooni. Paneme tähele, et vastastikrekursiooni kasutati ainult selleks, et igal rekursioonisammul oleks teada, kas jooksev element tuleb lisada ehitatava paari esimesse või teise komponenti. Seda infot saaks hoida lisaparameetris, järjehoidjas. Kuna on ainult kaks valikut, siis sobib tõeväärtustüüpi järjehoidja. Lisaparameetri sissetoomiseks kirjutame lokaalse abifunktsiooni. Nii saame koodi

```
kaheksLoe xs
  = let
      kaheksLoe b (x : xs)
        = let
            (us , vs)
              = kaheksLoe (not b) xs
          in
          if b then (us , x : vs) else (x : us , vs)
        kaheksLoe _ _
          = ([] , [])
    in
    kaheksLoe False xs
```
. (72)

Definitsiooni (72) lokaalse `kaheksLoe` esimene argument `b` on järjehoidja. Tema väärtuse False korral lisatakse jooksev element vasakusse listi, väärtuse True korral paremasse listi. Pangem tähele, kuidas igal rekursiivsel pöördumisel muudetakse tõeväärtusparameetri väärtus vastupidiseks. See asendab funktsioonide vahel hüppamist vastastikrekursiooni puhul ja paari komponentide vahetust definitsiooni (62) puhul.

Olgu meil nüüd vaja defineerida muutuja `maxArv` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks tema maksimaalsete elementide arvu. Hoolimata sarnasusest muutuja `nullideArv` spetsifikatsiooniga, mille realiseeris otserekursiivne kood (54), ei ole siin nii lihtne definitsioon võimalik. Põhjus on selles, et listi maksimaalne element pole eelnevalt teada, see tuleb alles töö käigus kindlaks teha, mistõttu pole näiteks esimese elemendi juurde asudes võimalik kohe otsustada, kas ta tuleb loendada või mitte. Muidugi on võimalik kõigepealt vaadata järele, milline element on maksimaalne, ja seejärel lugeda nad kokku, kuid see tähendaks listi kahekordset läbivaatamist.

Järjehoidjate kasutamine võimaldab piirduda listi ühekordse läbivaatusega. Sobib kood

```
maxArv
  :: (Ord a)
  => [a] -> Int
maxArv (x : xs)
  = let
      maxArv m i (z : zs)
        = case compare m z of
            GT
              -> maxArv m i zs
            EQ                          ,                    (73)
              -> maxArv m (i + 1) zs
            _
              -> maxArv z 1 zs
      maxArv _ i _
        = i
    in
    maxArv x 1 xs
maxArv _
  = 0
```

kus lokaalselt sissetoodud funktsioonil on kaks järjehoidjat `m` ja `i`. Järjehoidjas `m` on jooksev maksimaalne element, järjehoidjas `i` nende jooksev arv. Järjehoidja `i` kujutab endast loendurit. Algul on maksimaalne kogu listi esimene element, sest kaugemale pole vaadatud, ja nende arv on 1. Kui jooksev element on jooksvast maksimumist väiksem, siis parameetrite väärtused ei muutu; kui jooksev element võrdub jooksva maksimumiga, siis maksimaalsete arv tõuseb 1 võrra; kui aga jooksev element on jooksvast maksimumist suurem, siis muutub maksimum ja loendur läheb tagasi 1-ks. Kui kogu list on läbi vaadatud, antakse loendur välja.

Lokaalse `maxArv` definitsioon koodis (73) on näide nn sabarekursioonist. Sabarekursiivseks (ingl *tail recursion*) nimetatakse rekursiivset definitsiooni, mille järgi arvutades toimuvad kõik rekursiivsed pöördumised rekursioonitaseme viimase operatsioonina, rekursiivse pöördumise tulemusega ei tehta muud kui tagastatakse eelmisele tasemele. Lokaalse `maxArv` definitsioonis on rekursiivsed pöördumised ainult *case*-avaldise juhtude paremates pooltes ja kõigil juhtudel on pöördumise tulemus ühtlasi tagastatavaks väärtuseks. Võrdluseks näiteks (54) pole sabarekursiivne, sest kui argumentlisti pea on null, siis liidetakse rekursiivse pöördumise tulemusele veel 1.

Sabarekursiooni tähtsus seisneb selles, et selliseid definitsioone on võimalik transleerimisel optimeerida, sest kuna rekursiivse pöördumisega on jooksva rekursioonitaseme töö sisuliselt lõppenud, võib rekursiivsel pöördumisel jooksva taseme lokaalsete muutujate väärtused unustada või õigemini järgmise rekursioonitaseme väärtustega üle kirjutada. Nii tehes tekib funktsionaalsest rekursiivsest definitsioonist, nagu näiteks (73), imperatiivne tsükliga algoritm.

Viimase näitena järjehoidja kasutamisest defineerime muutuja `bilanss` väärtuseks funktsiooni, mis võtab argumendiks stringi ja annab tulemuseks tõeväärtuse vastavalt sellele, kas ümarsulgude bilanss selles klapib. Kasutame järjehoidjat, mis näitab asukoha sügavust sulgude suhtes ehk seda, mitmekordse sulupaari vahel parajasti ollakse. Sulgude bilanss klapib parajasti siis, kui stringi lõpus on sügavus 0 ja üheski kohas pole sügavus negatiivne. Sobib kood

```
bilanss
  :: String -> Bool
bilanss str
  = let
      bilanss n (c : cs)
        = let
            d = case c of
                  '('
                    -> 1
                  ')'
                    -> -1
                  _
                    -> 0
          in
          n >= 0 && bilanss (n + d) cs
      bilanss n _
        = n == 0
    in
    bilanss 0 str
```

järjehoidjaks on lisamuutuja `n`.

Vaatamata sellele, et lokaalse definitsiooni rekursiivne pöördumine on argumendipositsioonis, võib seda definitsiooni lugeda sabarekursiivseks, sest operaatori && teine argument väärtustatakse ainult juhul, kui esimese tõesus on juba kinnitust leidnud, ja sellisel juhul tagastab konjunktsiooni arvutus teise argumendi muutmata kujul. Kuna Haskellis on ka konstruktorid laisa väärtustamisega, on analoogselt võimalik õigustatult sabarekursiivseks lugeda ka definitsioonid (58) ja (60). Muud varasemad näitedefinitsioonid sabarekursiivsed ei ole.

Ülesandeid

217. Kirjutada definitsiooniga (71) samaväärne lihtsam definitsioon, kasutades vastastikrekursiooni asemel järjehoidjat.

218. Defineerida muutuja `maxKonst` väärtuseks funktsioon, mis võtab argumendiks võrdusega tüüpi elementide listi ja annab väärtuseks selle listi pikima konstantse (st võrdsete elementidega) lõigu pikkuse.

219. Selgitada, miks definitsioonid (49), (52), (72) pole sabarekursiivsed.

Akumulaatorid

Akumulaator (ingl *accumulator*) on rekursiivse funktsiooni argument, mis arvutuse käigus kogub endasse infot, akumuleerib seda. See tähendab, et rekursiivsel pöördumisel lisatakse sinna argumenti midagi juurde võrreldes tema varasema väärtusega.

Akumulaatoreid võib kasutada üsna mitmel eesmärgil. Mõnikord lihtsalt ei saa ilma läbi, nii nagu järjehoidjatetagi. Mõnikord aitab akumulaatorite kasutamine märgatavalt arvutuse aja- ja mälukulu kokku hoida. Arvutuse üleviimisega akumulaatoritesse võib definitsioon muutuda sabarekursiivseks. Osa programmeerijaid võib akumulaatoreid kasutada lihtsalt selleks, et järgida imperatiivse programmeerimise stiili, kus arvutuse lõpptulemus kujuneb tihti välja mingi perioodiliselt uuendatud väärtusega muutujas. Funktsionaalses keeles näeb imperatiivset paradigmat ahviv programm muidugi välja üsna teistmoodi kui imperatiivses keeles, taoline stiil koodi loetavust vaevalt et parandab.

Akumulaator on hädavajalik, kui on vaja potentsiaalselt lõpmatul listil otsida sobivas suhtes elemente. Olgu meil näiteks vaja mistahes listi kohta kontrollida, ega temas ei ole korduvaid elemente, ja saada selline kontrollifunktsioon muutuja `kordusteta` väärtuseks, mis rahuldab signatuuri

```
kordusteta
  :: (Eq a)     .
  => [a] -> Bool
```

Akumulaatoreid kasutamata saaks programmeerija kirjutada definitsiooni

```
kordusteta (x : xs)
  = not (elem x xs) && kordusteta xs
kordusteta _                          .
  = True
```
(74)

Kui argumentlist on tühi, tuleb väärtuseks True, mis on õige, sest tühjas listis ükski element ei kordu. Mittetühja listi puhul peab elementide mittekorduvuseks olema täidetud parajasti kaks tingimust: esimene element ei tohi esineda järgmiste hulgas — seda kontrollib parajasti koodiosa `not (elem x xs)` — ja nende järgmiste elementide hulgas ei esine kordumisi — seda kontrollib koodiosa `kordusteta xs` rekursiivselt. Paraku lõpmatul listil, milles esineb kordusi, kuid esimene element on unikaalne, selle definitsiooni järgi arvutus kordusi üles ei leia, sest püüab kõigepealt esimest elementi kõigi järgnevatega võrrelda.

Algoritm, mis siinkohal aitaks, peaks tegema võrdlusi teises järjekorras. Iga elementi tuleks võrrelda temast eespool olevatega, mitte tagapool olevatega nagu teeb (74). Igale elemendile eelneb listis vaid lõplik arv elemente ja seetõttu jõuaks niisugune algoritm suvalise kahe elemendi võrdlemiseni lõpliku arvu võrdluste järel.

Definitsioon

```
kordusteta xs
  = let
      kordusteta as (z : zs)
        = not (elem z as) && kordusteta (z : as) zs
      kordusteta _  _
        = True
    in
    kordusteta [] xs
```
(75)

realiseeribki korduvuste leidmise sellise algoritmiga. Lokaalselt defineeritud `kordusteta` akumulaatoris `as` hoitakse jooksvalt algse listi alguselemente, mille omavahelised võrdlused on juba tehtud. Järgnevat elementi võrreldakse parajasti kõigi akumuleeritud elementidega ning kui ta osutub neist kõigist erinevaks, lisatakse temagi seejärel akumulaatorisse. Arvutus selle definitsiooni järgi leiab korduse üles isegi juhul, kui list on osaline.

Pangem tähele, et argumentlisti elemendid salvestuvad akumulaatorisse algsega vastupidises järjekorras — esimene läheb viimaseks jne. Sel pole antud ülesande juures tähtsust, kuna akumulaatorit kasutatakse vaid kontrolliks, kas ta sisaldab mingit elementi.

Akumulaatori ja järjehoidja vahel pole kindlat piiri. Koodi (75) akumulaatoril on järjehoidja funktsioon, kuna peab meeles järge, kui kaugele on elementide suhted läbi kontrollitud. Samas võib igasuguseid loendureid lugeda akumulaatoreiks, sest rekursiivsetel pöördumistel lisatakse neisse midagi juurde.

Ülesandeid

220. Definitsiooniga (75) defineeritud funktsioon `kordusteta` kutsutakse välja argumendil, mille väärtus on $1:2:3:4:5:[]$, ja arvutatakse väärtus. Argumentide `z` ja `as` milliste väärtustega kutsutakse selle töö käigus välja `elem`?

221. Kui definitsioonis (75) konjunktsiooni pooled ära vahetada, kas definitsioon muutuks halvemaks, paremaks või pole vahet?

222. Cantori tolm 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 tolmu hulka või mitte.

223. Defineerida muutuja `disjunktsed` väärtuseks *curried*-kujuline funktsioon, mis võtab argumendiks kaks listi ja kui nad on lõplikud või lõpmatud ja sisaldavad ühiseid elemente, siis annab välja tõeväärtuse `False`, kui aga mõlemad listid on lõplikud ja ühiseid elemente ei sisalda, annab välja `True`. Kuidas see funktsioon töötab, kui üks listidest on osaline?

224. 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.

Akumulaatori üks tüüprolle on arvutuse vahetulemuste salvestamine. Sellisel juhul antakse lõpus välja kas akumulaator ise või rakendatakse talle enne veel mingit operatsiooni.

Vaatame näiteks, kuidas võiks programmeerida listi ümberpööramise. Vastav funktsioon on moodulis Prelude muutuja `reverse` väärtuseks; defineerime siin ta muutujasse `tagurpidi`, signatuur on

```
tagurpidi
  :: [a] -> [a]
```
.

Otserekursiivselt tehes saame definitsiooni

```
tagurpidi (x : xs)
  = tagurpidi xs ++ [x]
tagurpidi _
  = []
```
. (76)

Siin rekursiooni igal vahesammul tehakse rekursiivne pöördumine listi sabal ja selle tulemus konkateneeritakse listi pea ette. See annab kokku ruutkeerukusega arvutuse, sest kui tähistada n-ga argumentlisti pikkust, siis protsessi käigus vasakult konkateneeritavate listide pikkused on $0, 1, \ldots, n-1$, mille summa on ruutpolünoom n suhtes.

Seepärast on siin väga oluline kasutada akumulaatorit. Vaatame definitsiooni

```
tagurpidi xs
  = let
      tagurpidi (x : xs) as
        = tagurpidi xs (x : as)
      tagurpidi _        as
        = as
    in
    tagurpidi xs []
```
; (77)

selle järgi arvutades kirjutatakse elemendid ükshaaval järjest akumulaatorisse `as`. Arvutus on listi pikkuse suhtes lineaarse keerukusega, sest igal rekursioonisammul tõstetakse ainult üks element ringi. Ja kuna elemendid salvestuvad akumulaatorisse algsega võrreldes vastupidises järjestuses, siis lõpuks ongi akumulaatori väärtuseks just ümberpööratud esialgne list.

Ülesandeid

225. Defineerime muutuja `sabad` koodiga

```
sabad
  :: [a] -> [[a]]
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.

226. 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.

Arvutamine akumulaatoris võib muidugi olla keerulisem kui mingi listi koostamine. Akumulaatoriga võib rekursiivsel väljakutsel teha mingi suuremamahulise operatsiooni; siis akumulaator kui infokoguja tähendab arvutusoperatsioonide kogujat.

Võtame näiteks listirekursiooni juures defineeritud muutuja `geom`, mille väärtuseks oli funktsioon, mis võtab argumendiks arvud a ja q ning annab välja geomeetrilise jada algelemendiga a teguriga q lõpmatu listina. Listirekursiooni mitte kasutav definitsioon (65) oli ebaefektiivne. Akumulaatori abil on võimalik lineaarse ajaga arvutus kätte saada ka ilma listirekursiooni kasutamata. Sobib deklaratsioon

```
geom a q
  = a : geom (a * q) q .
```
(78)

See põhineb tähelepanekul, et geomeetrilise jada osa alates teisest liikmest on geomeetriline jada sama teguriga. Seega piisab igal sammul kirjutada jooksev `a` listi ja rekursiivselt pöörduda, korrutades `a`-d `q`-ga. Siin lokaalseid definitsioone pole, akumulaatorina töötab ülesande püstitusest tulenev parameeter `a`.

Listide defineerimine akumulaatori abil on tihti lihtsam kui listirekursiooniga defineerimine. Olgu meil vaja koostada list, milles on järjest kõik arvud kujul $1^4 + \ldots + n^4$, kus n muutub üle kõigi naturaalarvude; defineerime ta muutuja `suuredSummad` väärtuseks. Võttes jooksva summa salvestamise jaoks kasutusele akumulaatori `a` ja liidetava järjekorranumbri jaoks järjehoidja

`i`, saame koodi

```
suuredSummad
  :: (Integral a)
  => [a]
suuredSummad
  = let                                                    .          (79)
      suuredSummad a i
        = a : suuredSummad (a + i ^ 4) (i + 1)
    in
    suuredSummad 0 1
```

Arvutamine definitsiooni (79) järgi toimub üsna sarnaselt imperatiivse programmeerimise tsüklile, järjehoidja `i` on sisuliselt tsükliindeks. Algselt on akumulaatori väärtus 0 ja tsükliindeksi väärtus 1. Igal rekursiivsel pöördumisel lisatakse akumulaatorisse jooksva tsükliindeksi 4. astme liitmine, tsükliindeksile aga liidetakse 1. Niimoodi tekivad akumulaatorisse 1^4 liitmine, 2^4 liitmine jne. Tulemuslist moodustub akumulaatori olekutest selle protsessi jooksul.

Et sama listirekursiooniga teha, tuleks kasutada abifunktsiooni, mille kaudu saaks avaldada listi saba terve listi kaudu. Siin pole see seos kõige lihtsam.

Deklaratsiooniga (49) defineeritud funktsiooni `suurSumma` saaks nüüd samaväärselt defineerida selle lõpmatu listi kaudu deklaratsiooniga

```
suurSumma n
  | n >= 0
    = suuredSummad !! n                                    .          (80)
  | otherwise
    = error "suurSumma: negatiivne liidetavate arv"
```

Definitsiooni (80) järgi arvutamine on esimesel korral pisut ebaefektiivsem kui varasema definitsiooni järgi. Vahe tuleb esiteks sellest, et andmestruktuuri moodustamisele kulub lisaressurss, teiseks sellest, et lõpus lisandub listist lugemine. Kui aga mingi arvutuse käigus on vaja arve kujul $1^4 + \ldots + n^4$ paljude eri n-de jaoks, tasub neist listi moodustamine kindlasti ära. Siis igal järgmisel korral on vaja arvutada ainult neid summasid, mida seni veel arvutatud ja listi salvestatud pole.

Niisama lihtne on defineerida mingi muutuja väärtuseks lõpmatu list kõigist Fibonacci arvudest.

Koodiks sobib

```
fibs
  :: (Integral a)
  => [a]
fibs
  = let                          .                    (81)
      fibs a b
        = a : fibs b (a + b)
    in
    fibs 0 1
```

Kuna järjekordse Fibonacci jada liikme arvutamisel pole vaja teada seda, mitmes liige see on, siis saab hakkama tsükliindeksita. Samas kuna iga elemendi arvutamiseks on vaja kaht eelmist, on lokaalses `fib` definitsioonis kaks akumulaatorit, mis tähistavad jada kaht jooksvat elementi. Neist esimene hoiab seda Fibonacci arvu, mis hetkel tuleks listi kirjutada, teine järgmist. Igal rekursioonisammul nihkub teine argument esimese kohale ja uueks teiseks argumendiks tuleb argumentide summa. Nagu definitsioon (69), annab ka definitsioon (81) lineaarses ajas arvutuse.

Ülesandeid

227. Anda koodiga (70) defineeritud muutujale `collatz` samaväärne definitsioon akumulaatori abil.

228. Defineerida muutuja `facts` väärtuseks list, mille elementideks on järjest kõigi naturaalarvude faktoriaalid. Listi algusosa väljaarvutamine olgu lineaarse keerukusega arvutatud osa pikkuse suhtes.

229. Ülesandes 176 defineeriti Lucas' jada. Defineerida akumulaatorite abil muutuja `lucs` väärtuseks list, mille elementideks on järjest kõik Lucas' jada elemendid.

230. 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).

231. Otse, ilma ülesande 230 lahendust kasutamata defineerida muutuja `murdTäpsusega` väärtuseks *curried*-kujul funktsioon, mis võtab argumendiks ratsionaalarvu q ja täisarvu n ning annab väärtuseks arvu q kümnendesituse stringikujul, kus on kuni n kohta peale koma. Kui arvus on tegelikult rohkem kui n komajärgset kohta, siis tagumised kohad jätta ära ilma ümardamata. Kui arvus on vähem kui n komajärgset kohta, siis arvu lõppu nulle mitte kirjutada. Kui tulemusstringis komajärgseid kohti pole, siis jätta ära ka koma. Negatiivse n korral ei pea töötama.

232. Olgu $`$ paremassotsiatiivne binaarne operatsioon, mis defineeritakse reegliga

$$a ` r = a + \frac{1}{r}$$

(sama operatsioon defineeriti muutuja `//` väärtuseks koodiga (19)). Reaalarvu r **ahelmurruks** (ingl *continued fraction*) nimetatakse lõplikku või lõpmatut esitust vastavalt kujul $r = a_0 ` \ldots ` a_n$ või $r = a_0 ` a_1 ` \ldots$, kus kõik a_i on täisarvud ja ainult a_0 võib olla mittepositiivne, kusjuures lõplikkuse korral $a_n > 1$. Arve a_i nimetatakse ahelmurru **elementideks**.

Defineerida muutuja `ahel` väärtuseks funktsioon, mis võtab argumendiks ratsionaalarvu q ja annab välja q ahelmurru elementide listi.

Akumulaatoritega saab arvutada muidugi kõikvõimalikke asju, mitte ainult liste. Näiteks võime deklaratsiooni (79) eeskujul akumulaatoreid kasutades defineerida muutuja `suurSumma` otse, ilma listi vahenduseta. Sobib kood

```
suurSumma n
  | n >= 0
    = let
        suurSumma a i
          | i <= n
            = suurSumma (a + i ^ 4) (i + 1)
          | otherwise
            = a
      in
      suurSumma 0 1
  | otherwise
    = error "suurSumma: negatiivne liidetavate arv"
```

. (82)

Võrreldes definitsiooniga (79) on lisandunud väline parameeter `n` ja tema negatiivse väärtuse käsitlemine eraldi; listi koostamine on kadunud, kuid lisandunud on hargnemine lokaalses deklaratsioonis, kus juhul, kui tsükliindeksi väärtus ületab `n` oma, arvutus lõpetatakse ja antakse akumulaator välja. Arvutuspõhimõte on aga samasugune kui definitsiooni (79) puhul.

Muutuja `fib` saame ilma listi konstrueerimata defineerida koodiga

```
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)
```

. (83)

Siin on toimunud analoogilised muudatused nagu `suurSumma` definitsiooni juures. Peale nende on lokaalsesse definitsiooni lisandunud üks parameeter — tsükliindeks. Nüüd on teda vaja, sest töö tuleb pärast õiget arvu samme lõpetada. Tsükliindeksi muutumise suund on siin vastupidine definitsiooniga (82); selline valik on tehtud vaid programmeerimise mugavuse pärast.

Arvutus definitsiooni (83) käib jällegi lineaarse keerukusega argumendi suhtes. Definitsiooni (83) järgi toimub Fibonacci arvu leidmine isegi pisut kiiremini kui definitsiooni (53) järgi, sest definitsiooni (53) puhul konstrueeritakse igal rekursioonisammul paar, mis nõuab lisaressurssi. Koodi (83) eelis arvutuskiiruses koodiga (53) võrreldes on siiski vaid marginaalne.

Ülesandeid

233. Anda samaväärne definitsioon koodiga (51) defineeritud muutujale `kahFact`, mille järgi toimuks arvutamine akumulaatoris. Saada läbi kahe parameetriga.

234. Kirjutada ülesande 176 uus lahendus, mille korral arvutus toimub akumulaatorites ja vahestruktuure ei moodustata.

Väärtustusjärjekorra ohje

Võrreldes muutuja `suurSumma` algset definitsiooni (49) ja akumulaatoriga definitsiooni (82), märkame, et algse definitsiooni järgi arvutamisel ei saa liitmistehet sooritada enne, kui rekursiivne pöördumine on andnud tulemuse, samas kui akumulaatoriga arvutamisel oleks võimalik liitmine ära teha ilma rekursiivselt pöördumata.

Pika rekursiivsete pöördumiste jada kontekstis nähtub sellest, et definitsiooni (49) puhul ei sooritata ühtki liitmist enne, kui rekursioon on jõudnud põhjani, kõik liitmised saab teha alles rekursioonisügavusest tagasipöördumisel. See tähendab, et rekursiivsete pöördumiste jada enne baasjuhuni jõudmist kasvatab mällu pikka väärtustamata avaldist. Akumulaatoris arvutamine aga võimaldab liitmised sooritada juba “minnes” ja kui nii teha, siis mälutarve väheneb järsult.

See vahe on põhimõtteline. Kuigi Haskell väärtustab vaikimisi laisalt, mistõttu ka definitsiooni (82) puhul akumulaatoris liitmisi enne ei tehta kui alles lõpus ning mäluhõive on seetõttu niisama suur kui definitsiooni (49) puhul, annab akumulaator põhimõttelise võimaluse liitmisi jooksvalt sooritada, see tuleb vaid kuidagi ära kasutada. Definitsioon (49) aga tekitab pika väärtustamata avaldise paratamatult, agar väärtustamine laisa asemel ei muudaks midagi.

Haskellis on võimalik väärtustamist kohati agaraks pöörata, võttes appi agara rakendamise operaatori `$!`. Akumulaatorite kasutamine koos nende jooksva väärtustamisega annab olulise mälu kokkuhoiu.

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 $\bot$. Kujul f `$!` $\mathfrak{e}$ oleva avaldise arvutamisel väärtustatakse kõigepealt avaldist $\mathfrak{e}$ senikaua, kuni saadakse selline avaldis $\mathfrak{e}'$, mille kuju näitab, et väärtus on normaalne, misjärel väärtustatakse f $\mathfrak{e}'$. Pangem tähele, et avaldist $\mathfrak{e}$ ei väärtustata tingimata lõpuni, väärtuse normaalsuse selgumiseks piisab välimise konstruktori ilmumisest.

Näiteks avaldise `const 5 $! undefined` väärtus on $\bot$, tema väärtustamine lõpeb veateatega, samas kui avaldiste `const 5 $! 0` ja `const 5 $! (undefined , undefined)` väärtus on 5. Avaldise `const 5 $! length [1 .. ]` väärtustamine aga jääb lõpmatusse tsüklisse.

Operaatori `$!` erinevust operaatorist `$` näitab väga ilmekalt avaldiste

```
error "fun" $ error "arg"
```
(84)

ja

```
error "fun" $! error "arg"
```
(85)

väärtustamine. Kui (84) väärtustamisel tuleb välja funktsiooni viga, siis (85) väärtustamisel argumendi viga.

Meil on vaja definitsiooni (82) akumulaatorit jooksvalt väärtustama sundida. Selleks tuleb lisada operaator `$!` koodi sellise koha peale, et tema teine argument oleks see akumulaator. Siin osutub mugavaks prefiksne rakendamine, mille korral piisab **`($!)`** lisamisest rekursiivse pöördumise ette. Saame

```
suurSumma n
  | n >= 0
    = let
        suurSumma a i
          | i <= n
            = ($!) suurSumma (a + i ^ 4) (i + 1)
          | otherwise
            = a
      in
      suurSumma 0 1
  | otherwise
    = error "suurSumma: negatiivne liidetavate arv"
```
(86)

Tänu operaatorile `$!` väärtustab arvutusprotsess iga kord enne `suurSumma` rekursiivset väljakutset avaldise `a + i ^ 4`, seega uuendatud akumulaator antakse igal rekursiivsel väljakutsel edasi väärtustatud kujul, mis võtab vähem mälu. Definitsiooni (86) korral saab muutujat `suurSumma` edukalt kasutada mitu suurusjärku suuremate argumentidega kui eelmiste definitsioonide puhul.

Avaldist `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.

Muutuja `fib` defineeriva koodi (83) võime samasuguse lisandusega paremaks teha, saame

```
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)
```
. (87)

Definitsioonis (87) 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 plahvatust ei toimu.

Ülesandeid

235. Kirjutada oma moodulisse deklaratsioonid (49), (82), (86) ja neid kahekaupa välja kommenteerides kontrollida igaühe puhul, kui suure argumendiga on Hugs võimeline funktsiooni väärtust arvutama ilma mälu ületäitumiseta.

236. Kirjutada definitsiooni (87) teisend, mille korral ei jäetaks väärtustamata isegi mitte ühte liitmistehet.

237. Teisendada definitsioone (78) ja (79) nii, et listi komponentide väärtustamine käiks jooksvalt.

238. Modifitseerida definitsiooni (58) selliselt, et `fibs` arvutamisel definitsiooni (69) järgi toimuks jooksev summade arvutamine.

239. Kirjutada ülesannete 233 ja 234 lahendused nii, et arvutamise käigus pikki väärtustamata avaldisi ei tekiks.

240. Defineerida muutuja `synna` väärtuseks *curried*-kujul funktsioon, mis võtab täisarvulised argumendid n ja k ning annab tulemuseks tõenäosuse, et kui teha k sõltumatut valikut samast n elemendist, siis kõik k valitud väärtust on paarikaupa erinevad. Näiteks

`synna 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). Arvutus peab andma veateate, kui argumendid n ja k on sellised, mille korral antud ülesandepüstitus on mõttetu, veateade peaks võimalikult täpselt kirjeldama olukorda ja milles seisneb viga.

"Jaga ja valitse" tehnika

"Jaga ja valitse" (ingl *divide and conquer*) tehnika puhul jagatakse mittetriviaalne ülesanne igal rekursioonisammul kaheks või enamaks sama tüüpi alamülesandeks, mis lahendatakse rekursiivselt samal põhimõttel ja mille lahendustest kombineeritakse kokku terve ülesande lahendus. Triviaalsed alamülesanded lahendatakse otse, ilma jagamata.

Mõnikord tuleneb selline lahendusviis loomuldasa ülesande püstitusest, mõnikord aga on see taktika otstarbekas efektiivsuskaalutlustel, kuna ta võimaldab ühel või teisel põhjusel ressursse säästa. Kuid "jaga ja valitse" tehnikaga lahenduse keerukus sõltub paljudest asjaoludest ja alati selline arvutus kõige efektiivsem ei ole.

Järgnevalt on "jaga ja valitse" tehnika klassifitseerimine tehtud selle järgi, kas alamülesannetest, milleks ülesanne jaotatakse, on üks või rohkem mittetriviaalsed. Ainult ühe mittetriviaalse alamülesande puhul sarnaneb programmeerimine seninähtuga, sest samal rekursioonitasemel toimub igal juhul ülimalt üks rekursiivne pöördumine. Koodi koostamisel on aga vaja "jaga ja valitse" tüüpi mõtlemist. Kui mittetriviaalseid alamülesandeid on rohkem, siis võib samal rekursioonitasemel toimuda mitu rekursiivset pöördumist. Kogemus Fibonacci arvude arvutamisest definitsiooniga (52) näitab, et sellisel juhul võib arvutus väga ebaefektiivne olla, nii et tasub otsida optimeerimisvõimalusi.

Lõpuks vaatleme üht "jaga ja valitse" tehnika eripärast realiseerimisviisi.

Ühe mittetriviaalse alamülesandega olukorrad

Tüüpnäited "jaga ja valitse" tehnika kasutamisest, kus alati vaid üks alamülesannetest on mittetriviaalne, on igasugused kahendotsingud. Kahendotsingu (ingl *binary search*) puhul jagatakse otsinguruum kaheks mingis mõttes võrdseks osaks nii, et otsitav objekt saab olla ainult ühes neist, ja edasi toimitakse samamoodi selles osas. Niisuguse protsessi keerukus on logaritmiline algse otsinguruumi suuruse suhtes. Kahendotsingut, kus otsinguruum moodustab lõigu arvteljel ja igal sammul jagatakse lõik enam-vähem keskelt pooleks, nimetatakse lõigu poolitamise meetodiks.

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. Kuna ülipikkade arvude esitus ujukomaga on nii ebatäpne, et antud küsimus muutub mõttetuks,

jäävad kõrvale variandid muutuja `sqrt` kasutamisega. Kuna aga ruutfunktsioon on monotoonne, on võimalik kasutada lõigu poolitamise meetodit.

Selguse mõttes kodeerime kahendotsimise protsessi põhifunktsiooni definitsioonist eraldi. Põhifunktsiooni kood

```
täisruut
  :: Integer -> Bool
täisruut n
  | n > 1                                                    (88)
    = lähenda n (algpiirid n)
  | otherwise
    = n >= 0
```

pöördub muutujate `lähenda` ja `algpiirid` poole. Neist esimese rakendamine peab realiseerima otsinguprotsessi, teise rakendamine aga andma kätte lõigu, millest otsing peale võiks hakata.

Nende muutujate definitsioonid järgnevad all. Nagu definitsioonist (88) näha, kasutatakse kahendotsimist ainult juhul, kui argument on 1-st suurem. Muidu saab vastuse kohe välja kirjutada: argument on täisruut parajasti siis, kui ta on mittenegatiivne.

Realiseerime lõigu poolitamise protsessi nii, et jooksva lõigu alumine otspunkt saab ise olla täpne ruutjuur, kuid ülemine otspunkt mitte. Kasutades deklaratsiooniga (33) defineeritud ruututõstmisfunktsiooni `sqr`, annab soovitava protsessi meile kood

```
lähenda
  :: Integer -> (Integer , Integer) -> Bool
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
```

Arvutus töötab üldjoontes järgmiselt. Kui lõigu pikkus on 1, siis on uuritav arv täisruut parajasti juhul, kui ta on alumise otspunkti ruut, sest ülemine otspunkt on protsessi realisatsioonist tulenevalt ruutjuurest suurem. Kui lõigu pikkus on suurem kui 1, siis leitakse lõigu seest täisarv, mis on

võimalikult lõigu keskel, tehakse tema ruututõstmise teel kindlaks, kummale poole temast jääb uuritava arvu ruutjuur, ning pöördutakse rekursiivselt uue lõiguga. Võib ka juhtuda, et tema ongi ruutjuur — sellisel juhul on uuritav arv täisruut ja funktsioon annab välja True.

Igal rekursiivsel pöördumisel on tagatud, et uus otspunkt ei ole uuritava arvu täpne ruutjuur. Seega jääb kehtima invariant, et jooksva lõigu ülemine otspunkt ei ole täpne ruutjuur. Kui mingil rekursioonitasemel on lõigu pikkus 1-st suurem, siis järgmisel väljakutsel on lõik kindlasti lühem, mistõttu protsess varem või hiljem jõuab olukorda, kus lõigu pikkus on 1, ja lõpetab töö.

Jääb defineerida veel algset lõiku leidev muutuja `algpiirid`. Ka see peab respekteerima invarianti, et ülemine otspunkt peab olema uuritava arvu ruutjuurest suurem, alumine aga väiksem temast või võrdne temaga. Sobib näiteks kood

```
algpiirid
  :: Integer -> (Integer , Integer)
algpiirid n                                    .
  = (1 , n)
```
(89)

Kuna seda funktsiooni arvutatakse vaid 1-st suurema argumendiga, siis on argumendi ruutjuur tõepoolest argumendist väiksem ning 1 omakorda ruutjuurest väiksem.

Et alglõik sisaldab algparameetriga võrdse arvu arve, siis arvu n täisruuduks olemise kontroll seniste definitsioonide järgi on logaritmilise keerukusega n suhtes.

Alguslõiku saab aga ka kavalamalt määrata. On ju selge, et n on üldiselt liiga jäme hinnang arvule $\sqrt{n}$. Selle asemel võib kasutada ka algpiiride leidmiseks kahendotsingut. Alustame lõigust $[1;2]$ ning igal järgmisel sammul valime jooksvast kaks korda pikema lõigu, mille alumine otspunkt langeb kokku jooksva lõigu ülemise otspunktiga. Nii ilmuvad lõigud $[1;2]$, $[2;4]$, $[4;8]$ jne. Ruutjuure otsingu alglõiguks võtame selles protsessis esimese, mis hõlmab uuritava arvu ruutjuure oma sisepiirkonnas või alumise otspunktiga. Selle idee realiseerib definitsioon

```
algpiirid n
  = let
      piirid x y
        | sqr y > n
          = (x , y)                .
        | otherwise
          = piirid y (y + y)
    in
    piirid 1 2
```
(90)

On ilmne, et selles protsessis sobiva lõiguni jõudmiseks kuluvate sammude arv on logaritmiline lõpplõigu pikkuse suhtes. Kuna lõpplõigu pikkus ei ületa meie ruutjuurt, siis on nii see otsing kui ka järgnev lõigusisene otsing logaritmilise keerukusega selle ruutjuure suhtes. See on aga kokkuvõttes sama mis logaritmilisus algparameetri suhtes, nii et olulist ajavõitu me selle optimeeringuga ei saa.

Arvutust definitsiooni (90) järgi võib tinglikult nimetada lõigu poolitamiseks, sest vaadeldes kõigi otspunktide asemel nende pöördväärtusi, saame parajasti tavalise lõigu poolitamise protsessi, kus üheks otspunktiks on kogu aeg 0.

Ülesandeid

241. 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.

242. Defineerida muutuja `intSqrt` väärtuseks funktsioon, mis võtab argumendiks täisarvu n tüübist Integer: mittenegatiivse n korral annab tulemuseks n ruutjuure täisosa, negatiivse n puhul lõpetab töö spetsiaalse täitmisaegse veaga.

243. Defineerida muutuja `intLog` väärtuseks *curried*-kujul funktsioon, mis võtab argumentideks täisarvud a ja n: kui $\log_a n$ eksisteerib, siis annab väärtuseks $\log_a n$ täisosa, vastasel korral lõpetab spetsiaalse täitmisaegse veaga.

244. Defineerida muutuja `cosFix` väärtuseks võrrandi $x = \cos x$ võimalikult täpne ujukomaarvuga väljendatud lähend.

245. Kasutades ülesandes 182 defineeritud muutujat `kaheksPiiriJärgi`, defineerida "jaga ja valitse" tehnikaga muutuja `eraldaVäiksemad` väärtuseks *curried*-kujuline funktsioon, mis võtab argumendiks täisarvu n ja järjestusega tüüpi objektide listi l ja annab tulemuseks listipaari, milles l elemendid on ära jaotatud nii, et esimese listi ükski element pole suurem kui teise listi ükski element. Seejuures kui n on 0 ja l pikkuse vahel, siis esimeses listis on täpselt n elementi; kui n on 0-st väiksem, siis esimene list on tühi; kui n on l pikkusest suurem, siis on teine list tühi.

Mitme mittetriviaalse alamülesandega olukorrad

Vaatame siin tuntud Hanoi tornide ülesannet, mida imperatiivse keele baasil programmeerimise õpetamisel kasutatakse tihti näitena rekursiooni kasulikkusest. Hanoi tornide ülesanne seisneb järgmises. On antud n erineva läbimõõduga ketast, igaühel auk keskel, ja kolm püstist varrast, millest igaüks mahub ketta august läbi. Alguses on kõik kettad esimese varda peal alt üles suuruse kahanemise järjekorras. Ühe sammuga on lubatud tõsta suvalise varda pealt väikseim ketas mõne teise varda peale, millel pole temast väiksemaid kettaid. Ülesandeks on vähima arvu sammudega saavutada olukord, kus kõik kettad on teise varda peal.

Lahendus põhineb tähelepanekul, 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. 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.

Käsitleme lahendusena algul ümbertõstmiste jada. Loeme varraste nimedeks sümbolid A, B, C ja märgime üht ümbertõstmist kahetähelise stringiga, mille esimene täht näitab varrast, millelt väikseim ketas tuleb võtta, ning teine täht varrast, kuhu ketas tuleb panna. Sellega on tõstmine üheselt määratud. Defineerime muutuja `tõsted` väärtuseks funktsiooni, mis võtab argumendiks täisarvu n ja annab tulemuseks listi vajalikest ümbertõstmistest nende tegemise järjekorras.

Võttes appi vardaid tähistavad kolm lisaparameetrit, saame esimese lähendusena koodi

```
tõsted
  :: Int -> [String]
tõsted n
  | n >= 0
    = let
        tõsted 0 _ _ _
          = []
        tõsted n x y z
          = let
              r = n - 1
            in
            tõsted r x z y ++ [x, y] : tõsted r z y x
      in
      tõsted n 'A' 'B' 'C'
  | otherwise
    = error "tõsted: negatiivne argument"
```
. (91)

Abifunktsiooni kood 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 `tõsted 2`, saame vastuseks listi "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.

Paneme koodi (91) juures tähele, et list konstrueeritakse tsüklilise konkateneerimisega, kusjuures operaatori `++` vasakus argumendis on rekursiivne pöördumine. Kuid `++` vasak argument teatavasti kopeeritakse. See tähendab, et tekib mitmekordne kopeerimine: kuna rekursiivse pöördumise tulemus arvutatakse samamoodi, siis juba seal toimub kopeerimine jne. Esimesel rekursioonitasemel kopeeritakse pool listi, teisel tasemel pool sellest poolest ehk veerand listi, kolmandal kaheksandik jne. Vasakpoolseim element kopeeritakse igal rekursioonitasemel, neid on aga kokku samapalju kui kettaid. Seega tehakse tühiümbertõstmisi kokku $\frac{1}{2} + \frac{1}{4} + \ldots + \frac{1}{2^n}$ listipikkuse ehk umbes terve loodava listi jagu.

Et vältida ebaefektiivsust, mille operaatori ++ vasakus argumendis olev rekursiivne pöördumine kaasa toob, kirjutame uue definitsiooni, kus käiku läheb sama nõks mis listide ümberpööramise definitsiooni (77) juures: koostame listi akumulaatoris. Saame koodi

```
tõsted n
  | n >= 0
    = let
        tõsted 0 _ _ _ as
          = as
        tõsted n x y z as
          = let
              r = n - 1
            in
            tõsted r x z y ([x, y] : tõsted r z y x as)
      in
      tõsted n 'A' 'B' 'C'
  | otherwise
    = error "tõsted: negatiivne argument"
```
. (92)

Koodi (92) korral tühiümbertõstmisi ei toimu, iga element satub kohe oma lõplikku asupaika.

Hanoi tornide ülesande püstitaja võib aga oodata lahenduseks operatsioonide jada asemel seisude jada. Kirjutame ka selle tarvis koodi. Esitame seisu listikolmikuna, kus kolmiku komponendid vastavad varrastele ja iga list koosneb parajasti vastava varda otsas olevate ketaste järjekorranumbritest. Kettaid nummerdame suuruse kasvamise järjekorras.

Jagame ülesande osadeks nii nagu enne. Teeme seekord ilma vardaid märkivate lisaargumentideta, selle asemel tõstame rekursiivse pöördumise tulemuses seisude komponendid ringi. Eristavate lisaargumentide puudumisel on alamülesannetele vastavad rekursiivsed pöördumised ühesugu-

sed ja saame hakkama ühega. Tuleb kood

```
type Seis
  = ([Int] , [Int] , [Int])
hanoi
  :: Int -> [Seis]
hanoi n
  = case compare n 0 of
      GT
        -> let
             alam
               = hanoi (n - 1)
           in
           [(n : xs , zs , ys) | (xs , ys , zs) <- alam] ++
           [(zs , n : ys , xs) | (xs , ys , zs) <- alam]
      EQ
        -> [([] , [] , [])]
      _
        -> error "hanoi: negatiivne argument"
```

.

Kuna Hanoi ketaste ülesande väljund on ketaste arvu suhtes eksponentsiaalse suurusega, siis käsitletud optimeeringud — lisaparameetrite kasutamine esimeses ja ühekordne rekursiivne pöördumine teises interpretatsioonis — ei paranda töö efektiivsust rohkem kui kaks korda. Sama ei saa öelda näiteks astendamise korral.

Olgu meil vaja programmeerida tavaline arvu astendamine täisarvuga, mis positiivsel astendajal defineeritaks korduva korrutamise kaudu, negatiivsel aga pöördarvu leidmise abil. Kandes matemaatilise definitsiooni naiivselt üle Haskelli, saaksime definitsiooni

```
aste
  :: (Fractional a, Integral b)
  => a -> b -> a
aste a n
  = case compare n 0 of
      GT
        -> aste a (n - 1) * a
      EQ
        -> 1
      _
        -> 1 / aste a (- n)
```

, (93)

mille järgi arvutus toimub astendaja suhtes lineaarse keerukusega. Võtame appi “jaga ja valitse” tehnika: grupeerime tegurid kahte võimalikult ühesuurusse rühma, arvutame korrutise kummagi

rühma jaoks ja kombineerime kogutulemuse. Sellele vastab definitsioon

```
aste a n
  = case compare n 0 of
      GT
        -> let
             (q , r)
               = divMod n 2
             z = aste a q
           in
           if r == 0 then z * z else z * z * a
      EQ
        -> 1
      _
        -> 1 / aste a (-n)
```
, (94)

mille järgi arvutamine toimub astendaja suhtes logaritmilise keerukusega.

Võtmesammuks efektiivsuse kolossaalse paranemise juurde on siin mitte "jaga ja valitse" tehnika iseenesest, vaid ühe rekursiivse pöördumise saavutamine kahe asemele. See vähendab korrutamiste arvu. Kui avaldist `aste a q` mitte siduda lokaalse muutujaga ja **`in`** järel olevas avaldises panna `z` asemele `aste a q`, oleks kogu ümbertöötamise vaev kasutu, korrutamiste arv võrduks vanaviisi astendaja väärtusega.

Ülesandeid

246. Lülitada Hugsis sisse reduktsioonide (arvutussammude) ja kasutatud mäluühikute loendur (`:s +s`) ning arvutada ratsionaalarvude suuri astmeid nii definitsiooniga (93), definitsiooniga (94) kui definitsiooniga, mille saame viimasest, kui rekursiivse pöördumise tulemust muutujaga ei seota ja `z` asemel on tingimusavaldises `aste a q`. Veenduda, et viimane nõuab niisama palju ressurssi kui definitsioon (93), samas kui (94) järgi arvutamine on tunduvalt kiirem ja piirdub üliväikese reduktsioonide arvuga.

247. Bababab 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 bababab sõnad, siis $u+v'$ ja $u+v^*$ on bababab 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 195).

3. Kõik sõnad on saadavad punktide 1 ja 2 abil.

Defineerida muutuja `babababi` väärtuseks funktsioon, mis võtab argumendiks stringi ja annab tulemuseks True või False vastavalt sellele, kas string on babababi keele sõna või ei.

248. Kasutades ülesandes 182 defineeritud muutujat `kaheksPiiriJärgi`, defineerida muutuja `järjestaKiir` väärtuseks funktsioon, mis võtab argumendiks järjestusega tüüpi elementide listi ja annab tulemuseks järjestatud listi samade elementidega. Järjestamine toimugu kiirmeetodil ning vältida tuleb tühiümbertõstmisi.

249. Kirjutada ülesande 248 lahenduse teisend, mille korral ühtlasi kaotatakse kordumised, st tulemuslistis esinevad kasvavas järjestuses kõik algse listi elemendid, igaüks üks kord.

250. Kirjutada ülesande 248 lahenduse teisend, mille korral on tulemuseks 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.

Kahendpuukujuline kokkuarvutus listidel

Klassikalisteks näideteks “jaga ja valitse” tehnikast on listi järjestamine kiir- ja põimimismeetodil. Põimimismeetodil järjestamise põhimõtteks on jagada list kaheks võimalikult kärmelt, kasutamata ühtki võrdlemist, järjestada kumbki osa samal meetodil ning põimida tulemused üheks järjestatud listiks. See meetod garanteerib arvutustöö keerukuse $O(n \log n)$ listi pikkuse n suhtes.

Kahe järjestatud listi põimimisega oleme juba tutvunud, seda realiseeris definitsioon (68). Kuid seal oli vaja ühesuguste elementide kordumist vältida, siin aga tuleks kõik koopiad alles hoida. Seepärast anname siin muutujale `põimi` uue definitsiooni

```
põimi xs@ (a : as) ys@ (b : bs)
  | a <= b
    = a : põimi as ys
  | otherwise
    = b : põimi xs bs          .
põimi xs           []
  = xs
põimi _            ys
  = ys
```

Nüüd saaksime põimimismeetodil järjestamise defineerida koodiga

```
järjestaPõim
  :: (Ord a)
  => [a] -> [a]
järjestaPõim xs@ (_ : _ : _)
  = let
      (us , vs)                                    .          (95)
        = kaheksLoe xs
    in
    põimi (järjestaPõim us) (järjestaPõim vs)
järjestaPõim xs
  = xs
```

Listi kaheksjagamiseks on kasutatud koodiga (61), (62) või (72) defineeritud muutujat `kaheksLoe`.

Esimene deklaratsioon definitsioonis (95) sobib juhul, kui listis on vähemalt kaks elementi. Kuna list jagatakse kaheks võimalikult võrdseks osaks, on mõlemad osad sellisel juhul algsest listist lühemad, nii on rekursioon ohutu. Kui listis on vähem kui kaks elementi, siis pooleks löömine ei annaks kaht lühemat listi, seega tuleb teisiti käituda. Kuna 0- ja 1-elemendilised listid on juba järjestatud, piisab sellisel juhul anda originaallist välja.

Põimimismeetodil järjestamine definitsiooniga (95) sooritab alati suurusjärgus $n \log n$ sammu, kus n on listi pikkus, sest kõigi ühepikkuste listide jagamine osadeks toimub ühtviisi. Kui aga list on juba järjestatud või vaid mõned elemendid on vales positsioonis, on see liiga ebaefektiivne, sest siis oleks võimalik järjestada ka lineaarse keerukusega.

Sama arvutusidee võib realiseerida ka teisiti, kandes kaheks hargnemiste puu, mis seniste "jaga ja valitse" stiilis definitsioonide korral moodustus rekursiivsetest pöördumistest, üle puukujuliseks kokkuarvutuseks lististruktuuri peal.

Täpsemalt seisneb mõte järgnevas. Kaheksjagamise protsessi võib lihtsalt ära jätta, sest seal midagi sisulist ei toimu. Lähtume jagamisprotsessi lõpptulemusest, milleks on algse listi maksimaalne hakitus väikesteks listijuppideks; paneme kõik need listijupid ühte listi. Seejärel hakkame neid listijuppe iteratiivselt kahekaupa põimima, kuni lõpuks on kõik elemendid ühes listis. Iteratiivse põimimise programmeerimisel peame olema tähelepanelikud, et see toimuks kahendpuukujuliselt, muidu saame ruutkeerukusega järjestamise.

Maksimaalseks hakituseks võib ülalrealiseeritud koodi eeskujul võtta olukorra, kus algse listi iga element on omaette listijupis. Seda on lihtne saavutada koodiga

```
hakid
  :: [a] -> [[a]]
hakid xs                .                              (96)
  = [[x] | x <- xs]
```

Juppide kahekaupa põimimise üle kogu listi annab definitsioon

```
põimiKahekaupa
  :: (Ord a)
  => [[a]] -> [[a]]
põimiKahekaupa (xs : ys : yss)      .
  = põimi xs ys : põimiKahekaupa yss
põimiKahekaupa xss
  = xss
```

Operaatori `põimiKahekaupa` üks rakendamine viib läbi kõik puu ühe taseme põimimised. Selle tulemusel jääb juppide arv umbes kaks korda väiksemaks, kuid need jupid sisaldavad jätkuvalt kõik algse listi elemendid. Kui seda operatsiooni sooritada korduvalt, siis jõutakse lõpuks olukorrani, kus on jäänud vaid üks jupp, milles sisalduvad kõik algse listi elemendid järjestatult. Operaatori `põimiKahekaupa` itereerimise annab kood

```
põimiPuu
  :: (Ord a)
  => [[a]] -> [[a]]
põimiPuu xss@ (_ : _ : _)        ,
  = põimiPuu (põimiKahekaupa xss)
põimiPuu xss
  = xss
```

itereerimine toimub akumulaatoris.

Muutuja `põimiPuu` rakenduse tulemus ei sobi siiski lõppvastuseks, sest kui algne list on mittetühi, siis asub järjestatud list vastuslistis elemendina, ta tuleb sealt lõpuks välja võtta. Seepärast kujuneb järjestamisoperaatori definitsiooniks

```
järjestaPõim xs
  = case põimiPuu (hakid xs) of
      zs : _                    .
        -> zs
      _
        -> []
```
(97)

Kuna saadud järjestamisprotsess toimub sisuliselt samamoodi kui eelmine, on keerukus endiselt suurusjärgus $n \log n$, kus n on listi pikkus, sõltumata listist.

Kuid nüüd järgneb puänt. Ilmselt on hakkimise tulemuse juures olulised parajasti kaks asja: et "hakid" sisaldaksid kokku parajasti algse listi kõik elemendid ja et iga "hakk" oleks omaette järjestatud (muidu ei tööta põimimine korrektselt). Pole oluline, et põimimise etapp algaks just üheelemendilistest juppidest. Me võime "hakkideks" samahästi 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, on kogu järjestamisprotsess triviaalne ja lineaarse keerukusega listi pikkuse suhtes (ainsa segmendi leidmiseks tuleb list korra läbi vaadata).

Seega meil on vaja vaid operatsiooni, mis leiaks listi järgi tema segmendid ja moodustaks neist listi. Kuid see on meil juba olemas, definitsioonidega (59) ja (60) muutuja `segmendid` väärtuseks antud. Niisiis piisab kood (96) asendada koodiga

```
hakid
  :: (Ord a)
  => [a] -> [[a]].
hakid xs
  = segmendid xs
```

Ülesandeid

251. Kirjutada põimimismeetodil järjestamine listi elementide kahendpuukujulise kokkuarvutamisega ümber nii, et arvutusprotsess ühtlasi kaotaks elementide kordused.

252. Kirjutada põimimismeetodil järjestamine listi elementide kahendpuukujulise kokkuarvutamisega ümber nii, et protsess moodustaks paaride listi, kus paaride esimesed komponendid on parajasti argumentlistis esinevad elemendid kasvavas järjekorras, igaüks ühe korra, ja vastavad teised komponendid näitavad elemendi esinemise kordsust argumentlistis.

253. Kasutades ülesande 204 lahendust, defineerida muutuja `olümpia` väärtuseks funktsioon, mis võtab argumendiks selliste protseduuride listi, mis edastavad tõeväärtuse, ja annab välja protseduuri, mis korraldab nende protseduuride vahel olümpiasüsteemis turniiri ja edastab võitja. Kui mõnel etapil on järele jäänud paaritu arv võistlejaid, siis ühe neist võib võistlemata järgmisse vooru lasta. Turniiri iga mäng seisneb selles, et kaks protseduuri edastavad kumbki oma tõeväärtuse: kui need on erinevad, siis tõese edastanu võidab, vastasel korral proovitakse samamoodi uuesti, niikaua kui saadakse erinevad tõeväärtused.

Abstraheerimisvõimalused

Kõrgemat järku funktsioonid

Eelnevas on korduvalt esinenud vihjeid, et Haskellis on võimalik kasutada kõrgemat järku funktsioone, st funktsioone, mis võtavad funktsioone argumendiks. Oleme mõnda neist ka 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 Haskelli peamooduli Prelude kaudu on kasutatavad kümned ja kümned eeldefineeritud kõrgemat järku funktsioonid, lisaks neile tulevad paljud ka teistest moodulitest. Programmeerijal on endal võimalik kõrgemat järku funktsioone lihtsasti defineerida. See ei nõua ühtegi lisateadmist võrreldes sellega, mis käesolevas materjalis on muutujate defineerimise kohta eespool juba ära toodud.

Kuna funktsiooniga programmeeritakse üldiselt teatavat käitumist, võimaldab funktsioon funktsiooni argumendina esitada teatavat käitumist parameetrilisena teise käitumise suhtes. Selliselt defineeritud funktsioonid on laiema kasutusega kui funktsioonid, mille argumentide hulgas funktsioone pole, ja võimaldavad vältida koodikordust (enam-vähem sama käitumise programmeerimist ikka ja jälle uuesti).

Paljud kõrgemat järku funktsioonid abstraheerivad teatavaid tüüpilisi rekursiooniskeeme. See tähendab, et osates neid funktsioone kasutada, pääseb programmeerija rekursiooniskeemi ilmutatud kirjapanekust, rekursiooniskeem on peidetud kõrgemat järku funktsiooni sisse. Näiteks võiks programmeerija iga ülesande puhul, kus on vaja mingit funktsiooni listi igale elemendile rakendada, realiseerida lahenduse omaette rekursiooniskeemiga, kuid operaatori `map` kasutamine päästab ta sellest vaevast.

Haskellis on kõrgemat järku funktsioonid enamasti ühtlasi polümorfsed, mis tähendab, et nad on kasutatavad paljude erinevate tüüpide jaoks, omavad lisaks väärtusparameetritele ka ilmutamata tüübiparameetreid. Näiteks `map` tüüp on `(a -> b) -> ([a] -> [b])`, mis sisaldab kaht tüübimuutujat; see tähendab, et muutujal `map` on omaette väärtus iga nende tüübimuutujate väärtustuse korral, ehk teisi sõnu, need tüübiparameetrid on sisuliselt lisaargumendid. Tüübiparameetrid aga mujal kui signatuurides ei väljendu, nad jäävad ilmutamata (näiteks kirjutades `map (== 'X')`, jääb märkimata, et esimene tüübiparameeter on Char ja teine Bool). Polü-

morfism teeb funktsioonid veelgi universaalsemaks.

Kui konkreetne ülesanne või alamülesanne lahendub mõne standardse universaalse funktsiooni otsese rakendusena, on halb stiil minna mööda võimalusest see ära kasutada. Väärtuste esitamine universaalsete funktsioonide rakendusena aitab lugejal koodist kiiremini aru saada, võrreldes sellega, kui see väärtus oleks defineeritud ilmutatud rekursiooniskeemiga.

Järgnevas tutvume kõigepealt eeldefineeritud muutujatega, mille väärtuseks kõrgemat järku funktsioon, ja näeme, kuidas võib üht ja sama kõrgemat järku funktsiooni esmapilgul väga erinevates situatsioonides kasutada. Lõpuks jõuame ka oma kõrgemat järku funktsioonide kirjeldamiseni. Kõik järgnevas jaotises õpitavad muutujad on kättesaadavad moodulist Prelude.

Tähtsamad eeldefineeritud kõrgemat järku funktsioonid

Alustame tutvust eeldefineeritud kõrgemat järku funktsioonidega kõige lihtsamatest, mille ülesandeks on funktsioonide argumente mingil viisil ümber paigutada või struktureerida või neid funktsioone mingis kindlas omavahelises suhtes kasutada.

Muutuja `flip` väärtuseks on funktsioon, mis võtab argumendiks *curried*-kujulise funktsiooni f ja annab tulemuseks samuti *curried*-kujulise 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. (Kompilaator tegelikult kirjutabki paremsektsioonid ümber `flip` kaudu, mis kajastub ka veateadetes. Tuum-Haskellis sektsioone ei ole.) Näiteks `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 *curried*-kuju ja järjendiargumendiga kuju vahel.

Näiteks `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 *curried*-kujuline. Seega avaldised `curry fst` ja `const` on ekvivalentsed, samuti avaldised `uncurry const` ja `fst`.

Kui funktsiooni rakendamine seisneb oma argumendile mingite funktsioonide järjestrakenda-

mises, kusjuures need funktsioonid sellest argumendist ei sõltu, siis on tegemist funktsioonide kompositsiooniga (ingl *composition*) (matemaatilise analüüsi terminoloogias liitfunktsiooniga).

Infiksoperaatori `.` väärtuseks ongi parajasti funktsioonide komponeerimine kui operatsioon kahel funktsioonil. Kuna tegemist on *curried*-kujuga, siis rangelt võttes saab see funktsioon 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. Siit võib teha tähelepaneku, et operaatori `.` väärtuseks on selline kõrgemat järku funktsioon, mis oma argumentfunktsioonile rakendades annab tulemuseks jälle kõrgemat järku funktsiooni. Muidugi on tüübikorrektsuseks tarvilik, et esimesena rakendatava funktsiooni väärtusetüüp võrduks teisena rakendatava funktsiooni argumenditüübiga.

Näiteks avaldise `(+ 2) . (* 5)` väärtuseks on funktsioon, mis igal oma argumendil arvutab väärtuse, korrutades argumenti 5-ga ja liites saadud tulemusele 2. Niisiis avaldise `((+ 2) . (* 5)) 7` väärtus on 37, eelnevas vaadeldud avaldis `uncurry (flip div)` on aga samaväärselt operaatoriga `.` ümber kirjutatav kujul `(uncurry . flip) div`.

Ülesandeid

254. Testida interaktiivse interpretaatori käsurealt funktsiooni `flip` erinevate mittekommutatiivsete argumentfunktsioonidega.

255. Veenduda interaktiivse interpretaatori käsurealt testides, et `curry fst` töötab nagu `const` ja `uncurry const` töötab nagu `fst`.

256. Lambdaavaldise `\ x y -> y` väärtuseks on *curried*-kujuline funktsioon, mis võtab järjest kaks argumenti ja annab välja neist teise. Kirjutada veel kaks põhimõtteliselt erinevat sellesama väärtusega avaldist, mis sisaldavad ülimalt 10 sümbolit.

257. Kirjutada võimalikult lühike avaldis, mille väärtuseks on funktsioon, mis arvutab väärtusi, võttes oma argumendist siinuse ja saadud tulemusest koosinuse. Testida interaktiivses keskkonnas.

258. Kirjutada kaks täisargumenteeritud (st mittefunktsioonilise väärtusega) tüübikorrektset avaldist, milles muutujad `flip` ja `uncurry` oleksid järjest rakendatud: ühel juhul ühes, teisel juhul teises järjekorras. Kummalgi juhul kirjutada seejärel avaldis samaväärselt ümber, lisades ka kompositsiooni.

259. Küsida interaktiivses keskkonnas muutujate `flip`, `curry`, `uncurry`, `$`, `$!`, `.` tüübid ja saada neist aru.

Järgmisena vaatame funktsioone, mille ülesandeks on oma argumentfunktsiooni itereerimine.

Muutuja `iterate` väärtuseks 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 väärtuse x argumendiks ja annab välja lõpmatu listi, mis koosneb väärtustest x, $f\,x$, $f(f\,x)$ jne.

Näiteks avaldise `iterate (* 2) 1` väärtuseks on lõpmatu list kõigi 2 astmetega, st $1{:}2{:}4{:}8{:}16{:}\ldots$.

Selliselt koostatud listidega oleme eespool juba kokku puutunud; varem defineerisime neid kas listirekursiooniga või akumulaatoritega. Muutujat `iterate` kasutades võiksime näiteks deklaratsiooniga (66) või (78) antud muutuja `geom` samaväärselt defineerida koodiga

```
geom a q
  = iterate (* q) a
```

Ka Fibonacci jada on analoogiliselt kirjeldatav, ainult et kuna iga element määratakse kahe, mitte ühe eelmise poolt, siis tuleb itereerida paaride peal ja pärast võtta neist esimesed komponendid. Seetõttu pole tegemist just kõige efektiivsema meetodiga Fibonacci jada arvutamiseks, ehkki keerukus on lineaarne. Koodiks tuleb

```
fibs
  = [fst p | p <- iterate (\ (a , b) -> (b , a + b)) (0 , 1)]
```
(98)

Paarid listis, mille iteratsiooniprotsess tekitab, koosnevad parajasti kahest järjestikusest Fibonacci arvust. Kui (a, b) on selline paar, siis järgmises paaris on esimene arv muidugi b, teine on Fibonacci jadas talle järgnev $a+b$. Sellest tuleb itereeritava funktsiooni kohale just `\ (a , b) -> (b , a + b)`. Esimesed kaks Fibonacci arvu on 0 ja 1, seetõttu on algpaari kohal `(0 , 1)`.

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 väärtuse 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

260. Kasutades erisüntaksi asemel muutujat `iterate`, kirjutada avaldis, mille väärtuseks on mingi aritmeetiline jada.

261. Kirjutada avaldis, mille väärtuseks on lõpmatu list, mille elemendid on järjest kõik ainult suurtest A-tähtedest koosnevad lõplikud stringid alustades tühjaga.

262. Anda ülesandele 211 uus lahendus kõrgemat järku funktsioonide abil.

263. Kasutades ülesandes 187 defineeritud muutujat `summad'`, kirjutada muutuja `pascal` väärtuseks lõpmatu list, mille element nr n on Pascali kolmnurga rida nr n listi kujul. Nii loodava listi elemente kui ka Pascali kolmnurga ridu loendame alates 0-st.

264. Arvutada interaktiivse interpretaatori käsurealt kirjutatud ühe avaldise abil esimene arvu 18 aste, mis jagub 486-ga.

265. Muutuja `until` abil defineerida muutuja `collatzPikkus` väärtuseks funktsioon, mis positiivsel täisarvulisel argumendil n annab tulemuseks elementide arvu n-ga algavas Collatzi jadas.

266. Küsida interaktiivses keskkonnas muutujate `iterate` ja `until` tüübid ja saada neist aru.

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 tulemusväärtuse.

Muutuja `takeWhile` väärtuseks 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 $1:2:4:8:16:32:64:[]$, sest need elemendid 2 astmete listis 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, et kuskil tagapool on ka predikaati rahuldavaid elemente, enam määravaks ei saa.

Muutuja `dropWhile` väärtuseks on funktsioon, mis võtab samad argumendid nagu eelmine, kuid annab neil tulemuseks argumentlisti selle osa, mis `takeWhile` puhul üle jääb. Niisiis 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.

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 mitterahuldamine esimeste elementide poolt ei sunni otsimist lõpetama. Avaldise `filter isUpper "Tere, Haskell!"` väärtus on "TH"; võrdluseks `takeWhile isUpper "Tere, Haskell!"` annab väärtuseks "T" ja `dropWhile isUpper "Tere, Haskell!"` annab "ere, Haskell!".

Operaatorite `takeWhile` ja `dropWhile` rakendamisel vaadatakse listi läbi ainult niikaugele, kuni leitakse predikaati mitte rahuldav element. Operaator `filter` vaatab aga alati listi kõik elemendid läbi. See võib valmistada üllatuse lõpmatu listi puhul, milles teatud kohast alates predikaat enam ühelgi elemendil ei kehti: siis jääb tulemuslisti arvutamine lõpmatusse tsüklisse, selle asemel et list normaalselt ära lõpetada. Näiteks `filter (<= 100) (iterate (* 2) 1)` väärtus on $1{:}2{:}4{:}8{:}16{:}32{:}64{:}\bot$, st osaline, mitte lõplik list.

Lisaks võiks mainida muutujaid `all` ja `any`, mis ka võtavad samasugused argumendid nagu `takeWhile`, `dropWhile` ja `filter`, kuid nemad annavad tulemuseks tõeväärtuse: `all` puhul tuleb True siis, kui listi iga element rahuldab predikaati ja False, kui mõni ei rahulda; `any` puhul tuleb True siis, kui listi mõni element rahuldab predikaati ja False, kui ükski ei rahulda. Väärtustamisel vaadatakse listi alates algusest senikaua, kuni leitakse element, mis vastavalt kas predikaati ei rahulda (`all` puhul) või rahuldab (`any` puhul).

Näiteks avaldise `all isSpace` väärtus on funktsioon, mis võtab argumendiks stringi ja annab tulemuseks True, kui see string koosneb ainult tühisümbolitest, ja False vastasel korral. Avaldise

```
filter (not . all isSpace)
```
(99)

väärtus on seega funktsioon, mis võtab argumendiks stringide listi l ja annab tulemuseks listi neist stringidest listis l, mis ei koosne ainult tühisümbolitest. Tõepoolest, antud predikaat on rahuldatud parajasti neil stringidel, mille kõigi sümbolite tühisuse kontrolli tulemusele eitust rakendades saame True.

Pisut keerulisema näitena kirjutame nüüd koodi, mis arvutab kõik algarvud. Osutub, et definitsioonike

```
algs
  :: [Integer]
algs
  = let
      isAlg n
        = all ((/= 0) . (n `mod`))
              (takeWhile ((<= n) . sqr) algs)
    in
    2 : filter isAlg [3 .. ]
```

annabki muutuja `algs` väärtuseks kõigi algarvude listi. (Liiga pika rea vältimiseks on avaldise `all ((/= 0) . (n `mod`))` argument `takeWhile ((<= n) . sqr) algs` viidud järgmisele reale; see on Haskelli koodipaigutusreeglite järgi lubatud.) Muutuja `sqr` peab olema eraldi defineeritud ruutfunktsioonina, näiteks koodiga (33).

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 jookseb, kuna iga järgmise arvu kontrollimisel läheb vaja ainult neid algarve, mis on selleks ajaks juba leitud.

Ülesandeid

267. Kirjutada funktsioon, mis argumenttekstil annab välja tema pikima väiketähti mitte sisaldava algusjupi.

268. Arvutada interaktiivse keskkonna käsureale kirjutatud ühe avaldise abil, mitu arvu 3 astet on ülimalt 100-kohalised.

269. Defineerida muutuja `elimTaanded` väärtuseks funktsioon, mis võtab argumendiks mingi teksti stringide listi kujul ja annab tulemuseks samal kujul teksti, kus stringide algusest on tühisümbolite jorud ära kaotatud.

270. Kirjutada avaldis (99) samaväärselt ümber, nii et `all` asemel oleks kasutatud `any`.

271. 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.

272. Defineerida muutuja `elimKorduvad` väärtuseks funktsioon, mis võtab argumendiks listi ja annab tulemuseks listi, milles esinevad parajasti kõik argumentlisti elemendid, kuid igaüks vaid ühe korra.

273. Kirjutada tüübikorrektne avaldis, kus muutuja `filter` argumendiks oleks `snd` ja järgmiseks argumendiks mõni mittetühi list.

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 `[x * x | x <- [1 .. 9]]` on samaväärselt ümber kirjutatav kujul `map (\ x -> x * x) [1 .. 9]`. Operaatori `map` kasutamisel on see eelis, et funktsioon, mis listile rakendatakse, on ilmutatud kujul alamavaldisena esitatud; toodud näites oli selleks `map (\ x -> x * x)`.

Operaatori `map` analoog, mis tegutseb korraga kahel listil, on `zipWith`. Tema väärtuseks on funktsioon, mis võtab järjest argumentideks mingi kahe argumendiga *curried*-kujulise 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 ülejääv osa unustatakse lihtsalt ära.

Näiteks `zipWith (*) [2, 3] [5, 7, 11]` väärtus on $10 : 21 : []$. Tulemuslisti esimene element 10 saadakse kui argumentlistide esimeste elementide 2 ja 5 korrutis, tulemuse teine element 21 saadakse kui argumentide teiste elementide 3 ja 7 korrutis ning et esimeses argumentlistis rohkem midagi pole, siis lõpeb siin ka tulemuslist.

Operaatori `zipWith` abil saab elegantselt realiseerida näiteks funktsioone, mille arvutamine nõuab listi kahe järjestikuse elemendi vaatlemist igal rekursioonisammul. Sellist funktsiooni otse defineerida on mõneti kohmakas, näiteks võiks tuua operaatori `summad` definitsiooni (58). Kuid `zipWith` abil on `summad` lihtsasti defineeritav koodiga

```
summad xs
  = zipWith (+) xs (tail xs)·
```
(100)

Et `tail xs` tulemuseks on argumentlisti nihe ühe komponendi võrra, siis listide vastavad elemendid, mis kokku liidetakse, ongi parajasti algses listis järjestikused.

Pangem tähele, et definitsioon (100) 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; kui esimene list on tühi, siis antakse kohe tühi list välja ja teine list jääb üldse uurimata.

Operaatorit `summad` läks vaja Fibonacci jada arvutamiseks listirekursiooni abil definitsiooniga (69). Asendades seal pöördumise funktsiooni `summad` poole definitsiooni (100) parema poole järgi, saame Fibonacci jadale uue definitsiooni

```
fibs
  = 0 : 1 : zipWith (+) fibs (tail fibs)'
```

mille põhimõte on sama mis koodil (69). Pisut teisendades saame elegantsema definitsiooni

```
fibs@ (_ : fs)
  = 0 : 1 : zipWith (+) fibs fs·
```
(101)

Ülesandeid

274. Küsida interaktiivses keskkonnas muutujate `map` ja `zipWith` tüübid ja saada neist aru.

275. Komprehensioonsüntaksit kasutamata arvutada string koodide järjekorras kõigist sümbolitest, mis kooditabelis leiduvad.

276. Komprehensioonsüntaksit kasutamata 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.

277. Kirjutada avaldis, mille väärtuseks on lõpmatu list lõpmatutest listidest. Anda see avaldis interaktiivses keskkonnas argumendiks sellisele avaldisele, mille tulemusena leitakse igast listist mõned elemendid.

278. Defineerida muutuja `nurk` väärtuseks *curried*-kujuline 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 169 lahenduseks olev avaldis, peab tulemuseks olema 10×10 korrutustabel.

279. Kirjutada definitsioon (100) samaväärselt ümber, võttes eeskuju definitsioonist (101): `tail` poole pöördumise asemel tuleb argumentlisti saba saada näidisesobitusest.

280. 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.

281. Lahendada kõrgemat järku funktsioonide abil ülesanded 185 ja 186.

282. Lahendada kõrgemat järku funktsioonide abil ülesanne 228.

283. 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.

284. Kui `zipWith` argumendi väärtus on kommutatiivne funktsioon, kas siis teeb alati sama välja, ükskõik kummas järjekorras anda listid?

Muutuja `foldl` väärtuseks on funktsioon, mis võtab argumentideks järjest binaarse operatsiooni $\oplus$, objekti e ja listi l ning annab välja objekti, mis tuleb lõppvastuseks, kui alustame väärtusest e ja igal sammul arvutame jooksva väärtuse paremalt operatsiooni $\oplus$ abil kokku listi l järjekordse elemendiga, läbides nii kogu listi elemendid suunaga algusest lõpu poole. See tähendab, et kui l elemendid on $a_1, \ldots, a_n$, siis on tulemuseks $(\ldots((e \oplus a_1) \oplus a_2) \oplus \ldots) \oplus a_n$. 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 $1:3:7:[]$.

Sarnaselt muutujaga `foldl` on muutuja `foldr` väärtuseks funktsioon, mis võtab argumentideks järjest operatsiooni $\oplus$, objekti e ja listi l ning annab välja objekti, mis tuleb lõppvastuseks, kui alustame väärtusest e ja igal sammul arvutame jooksva väärtuse vasakult operatsiooni $\oplus$ abil kokku listi l järjekordse elemendiga, läbides nii kogu listi elemendid suunaga lõpust alguse poole. See tähendab, et kui l elemendid on $a_1, \ldots, a_n$, siis on tulemuseks $a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus e)\ldots))$. 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 arvutatakse $1-(2-(3-0)) = 1-(2-3) = 1-(-1) = 2$.

On ilmne, et assotsiatiivse kommutatiivse operatsiooni (nagu liitmine) ja lõpliku listi puhul annavad `foldl` ja `foldr` alati sama tulemuse. Sama saab öelda juhul, kui operatsioon on assotsiatiivne ja mitte tingimata kommutatiivne, kuid e on selle operatsiooni ühikelement. Tõepoolest: listi elemendid on mõlemal juhul omavahel samas järjestuses ja ühikelemendi korral on ükspuha, kummas listi otsas tema asub. Näiteks ka avaldise `foldr (++) [] [[1, 3], [7], []]` väärtus on $1:3:7:[]$.

Pangem aga tähele, et isegi võrdse lõppväärtuse puhul ei pruugi olla ükskõik, kas kasutada operaatorit `foldl` või `foldr`. 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.

Operatsioon, millega kokkuarvutus toimub, ei pea üldjuhul olema sama tüüpi argumentidega. Nõutav on ainult see, et operatsiooni väärtusetüüp peab võrduma vastavalt kas esimese argumendi tüübiga (`foldl` puhul) või teise argumendi tüübiga (`foldr` puhul). Sellest piisab, et kokkuarvutuse käigus ei ilmneks tüübisobimatust.

Operaator `foldr` abstraheerib kõiki selliseid rekursiooniskeeme listidel, kus tulemus mittetühjal listil leitakse listi pea ja sama arvutuse tulemuse järgi listi sabal. See tuleb otse välja esitusest $a_1 \oplus (a_2 \oplus (\ldots \oplus (a_n \oplus e)\ldots))$, kus väärtus on väljendatud listi pea a_1 ja listi sabale vastava esituse $a_2 \oplus (\ldots \oplus (a_n \oplus e)\ldots)$ kaudu.

Selle illustreerimiseks anname koodiga (54) defineeritud muutujale `nullideArv` (väärtuseks listi nullide arvu leidev funktsioon) uue definitsiooni `foldr` kaudu. Sobib kood

```
nullideArv xs
  = let
      op x ülejäänu
        | x == 0
          = 1 + ülejäänu ,
        | otherwise
          = ülejäänu
    in
    foldr op 0 xs
```

milles muutujad `x` ja `ülejäänu` on kasutatud samas tähenduses nagu koodis (54).

Seevastu operaatori `foldl` abil saab programmeerida kõiki selliseid ühe akumulaatoriga arvutamisi, kus akumulaatori uus väärtus leitakse vanast, rakendades talle ja mingile elemendile antud operatsiooni, ja lõpus antakse akumulaator välja. Akumulaatoriga arvutamise esitamiseks operaatori `foldl` abil tuleb need elemendid, millega akumulaatorit kokku opereeritakse, paigutada järjekorras listi.

Näiteks koodiga (82) 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: negatiivne liidetavate arv"
```
. (102)

Samaväärsus ei tähenda siin ainult samu tulemusi samadel argumentidel, vaid ka sarnast arvutusskeemi: nii definitsiooni (82) kui ka (102) korral toimub arvutus akumuleerivalt.

Koodiga (77) defineeritud muutuja `tagurpidi` (väärtuseks listi ümberpööramine) võiksime samaväärselt anda koodiga

```
tagurpidi xs
  = foldl (flip (:)) [] xs
```

Muus osas `foldl` moodi töötava, kuid akumulaatori vaheväärtusi listi salvestava rekursiooniskeemi abstraktsioon on operaator `scanl`. Näiteks definitsiooniga (79) antud muutuja `suuredSummad` (väärtuseks lõpmatu list, kus positsioonil n on arvude 1 kuni n neljandate astmete summa) saab samaväärselt defineerida kujul

```
suuredSummad
  = scanl (\ a i -> a + i ^ 4) 0 [1 .. ]
```

Operaator `scanl` võimaldab defineerida Fibonacci arvude listi koodiga

```
fibs
  = 0 : scanl (+) 1 fibs'
```

mis on elegantsem kõigist senivaadelduist.

On olemas ka operaator `scanr`, mis vastab operaatorile `foldr` samamoodi nagu `scanl` operaatorile `foldl`.

Ülesandeid

285. Küsida interaktiivses keskkonnas muutujate `foldl`, `foldr`, `scanl`, `scanr` tüübid ja saada neist aru.

286. Anda koodiga (102) defineeritud muutujale `suurSumma` samaväärne definitsioon `foldr` kaudu.

287. Lahendada ülesanne 228 operaatori `scanl` abil.

288. Anda ülesandes 263 defineeritud muutujale `pascal` uus definitsioon ilma muutujata `summad'`, kasutades ülesandes 283 defineeritud muutujat `diagPööre`.

Erindid

Protseduuride programmeerimise juures õppisime, et iga protseduur tüüpi IO A edastab väärtuse tüübist A.

Täpsem olles tuleb nentida, et see on nii ainult juhul, kui protseduuri täitmine lõpeb normaalselt. Protseduur võib ka ebaõnnestuda — keskkonnaga suhtlemisel on see tavaline, nt kui püütakse lugeda faili, mida pole või mille lugemisõigus kasutajal puudub —, mispuhul väärtuse edastamise asemel protseduur hoopiski heidab (ingl *throw*) erindi. Erindid (ingl *exception*) on spetsiaalsed väärtused, mis tähistavad võimalikke erandolukordi.

Standard-Haskellis kuuluvad kõik erindid ühte tüüpi IOError, mille nimi viitab veale sisendi-väljundi käigus. See tähendab reaalselt seda, et mujal kui protseduurides tekkinud erindeid pole võimalik keeles väljendada ega siis ka töödelda. Haskelli laiendustes on olemas palju muid erinditüüpe. Üldse on Haskelli praegune standard erindikäsitluse poolest üsna algeline. Sellegipoolest piirdume siin Haskell 98 võimalustega, sest erinevate realisatsioonide isetegevuslikud laiendused ei pruugi olla kooskõlas ei omavahel ega ka tulevikustandarditega.

Esmajärjekorras läheb erinditöötlust vaja failidega opereerimisel. Moodulist Prelude saab 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 stringina. Muutujate `writeFile` ja `appendFile` väärtuseks on *curried*-kujuline funktsioon, mis võtab argumendiks failinime ja stringi ja annab tulemuseks protseduuri, mis kirjutab selle stringi antud nimega faili; seejuures kui fail on juba olemas, siis `writeFile` puhul tehakse fail enne tühjaks, `appendFile` puhul aga kirjutatakse string faili lõppu olemasoleva sisu järele.

Kõigil neil juhtudel heidab protseduur erindi, kui faili ei õnnestunud avada. (Moodulis Prelude on ainult kõige elementaarsemad failioperatsioonid. Keerulisemad on koondatud moodulitesse System.IO ja System.Directory.)

Kui *do*-avaldise mõne rea täitmine heidab erindi, siis lõpetab kogu *do*-avaldis kohe täitmise eba-

normaalselt, heites sama erindi. Vaatame näiteks koodi

```
lugemine
  :: IO ()
lugemine
  = do
      putStr "Anna loetava faili nimi. "
      fn <- getLine
      sisu <- readFile fn
      putStrLn ("Faili " ++ fn ++ " sisu:")
      putStrLn sisu
      putStrLn "(Lõpp.)"
```

Kui sellega defineeritud muutuja `lugemine` välja kutsuda, siis küsib ta faili nime. Kui sisestada eksisteeriva lugemisõigusega faili nimi (see satub muutujasse `fn`), siis loeb ta selle faili sisu (see satub muutujasse `sisu`) ja vormistab selle ekraanile. Kui aga faili avamine lugemiseks mingil põhjusel ebaõnnestub, näiteks faili puudumise tõttu, siis heidetakse faili lugemisel erind, mille tagajärjel kogu protseduur lõpetab töö veateatega.

Erindite püüdmiseks on operaator `catch`. Tema väärtuseks on *curried*-kujuline funktsioon, mis võtab argumendiks protseduuri a ja `püünise` (ingl ***handler***) h, mis kujutab endast funktsiooni, mille argumendiks on suvaline erind ja väärtuseks protseduur. Neil argumentidel on tulemuseks protseduur, mis täidab protseduuri a: kui see lõpetab normaalselt, siis muud ei juhtugi ja edastatakse sama väärtus, mille edastab a; kui aga a heidab erindi e, siis täidetakse lisaks protseduur $h\,e$, misjärel sõltuvalt tema õnnestumisest/ebaõnnestumisest kas edastatakse tema edastatud väärtus või heidetakse tema heidetud erind. Seega on tüübikorrektsuseks vaja, et püünis h annaks välja a-ga sama tüüpi protseduure (analoogiliselt sellega, miks tingimusavaldisel peavad harude tüübid kokku langema).

Näiteks võiksime oma faililugemiskoodi täiendada deklaratsioonidega

```
lugemineE
  :: IO ()
lugemineE
  = lugemine `catch` \ _ -> putStrLn "Lugemisel tekkis erind!"
```

Kutsudes nüüd välja muutuja `lugemineE`, toimub õnnestunud failiavamise puhul kõik nagu varem, ebaõnnestumise puhul aga pärast failinime sisestamist ilmub ekraanile kiri “Lugemisel tekkis erind!”.

Tavaliselt soovib kasutaja erindi tekkimisel saada täpsemat infot erindi iseloomu kohta. Näiteks faili lugemisel on kaks põhimõtteliselt erinevat võimalust erindi tekkimiseks: faili puudumine ja tema lugemisõiguse puudumine. Moodulist System.IO.Error tulevad muutujad, mille väärtuseks on sellised predikaadid, mis näitavad erindi iseloomu. Kasutades neid, võime kirjeldada

püünise, mille käitumine sõltub erindist. Sobib näiteks deklaratsioon

```
püünis
  :: IOError -> IO ()
püünis e
  | isDoesNotExistError e
    = putStrLn "Pole faili!"
  | isPermissionError e
    = putStrLn "Pole faili lugemisõigust!"
  | otherwise
    = putStrLn "Tekkis tundmatu erind."
```

Erinditöötlusega lugemise protseduuri kirjeldab nüüd eriti lihtne kood

```
lugemineE
  = lugemine `catch` püünis
```

Mõnikord 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.

Kirjutame näiteks meie püünise ümber kaheks erinevaks, millest üks püüab ainult õiguste puudumisest tekkivaid erindeid, teine ainult faili puudumisest tekkivaid. Probleem tekib nüüd sellega, et erindi läbilaskmine on rangelt võttes võimatu: püünis kui funktsioon võtab argumendiks suvalise erindi ja annab midagi tulemuseks. Kui kirjutaksime koodi

```
püüaÕigused
  :: IOError -> IO ()
püüaÕigused e
  | isPermissionError e
    = putStrLn "Pole faili lugemisõigust!"

püüaMuud
  :: IOError -> IO ()
püüaMuud e
  | isDoesNotExistError e
    = putStrLn "Pole faili!"
  | otherwise
    = putStrLn "Tekkis tundmatu erind."

lugemineE
  = lugemine `catch` püüaÕigused `catch` püüaMuud
```

siis `lugemineE` väljakutse ei töötaks enam nagu varem: faili puudumise korral satub vastav erind kõigepealt valesse püünisesse, kus hargnemiskonstruktsiooni sobiva juhu puudumise tõttu tekib näidisesobituse viga, st väärtus $\bot$, mida ükski püünis enam ei võta.

Lahendus seisneb selles, et töötlemisele mittekuuluvad erindid uuesti heita. Selleks on olemas muutuja `ioError`, mille väärtuseks on funktsioon, mis võtab argumendiks erindi e ja annab tulemuseks protseduuri, mis midagi ei tee, aga heidab erindi e. Seega operaator `ioError` on analoogne operaatoriga `return`: esimene on paljas erindiseade, teine paljas normaalse väärtuse seade. Teiselt poolt on `ioError` analoogne operaatoriga `error`, sest ta on kasutatav sõltumata tüübikontekstist.

Muutujate `püüaÕigused` ja `püüaMuud` õiged definitsioonid käiksid seega koodiga

```
püüaÕigused e
  | isPermissionError e
    = putStrLn "Pole faili lugemisõigust!"
  | otherwise
    = ioError e

püüaMuud e
  | isDoesNotExistError e
    = putStrLn "Pole faili!"
  | otherwise
    = putStrLn "Tekkis tundmatu erind."
```

Kui aga juba erindiseadeks läheb, siis võib olla mugav luua hoopis oma erindeid ning seada ja püüda neid. Oma erindi loomiseks on operaator `userError`. Tema väärtuseks on funktsioon, mis võtab argumendiks stringi ja annab tulemuseks erindi. Pangem tähele, et see funktsioon seda erindit ei sea; ta ei tegele sisendi-väljundiga üldse. Seadeks tuleb ikka kasutada operaatorit `ioError`.

Püüniste järgmine versioon, mis kasutab omatekitatud erindeid, võiks olla

```
püüaÕigused e
  | isPermissionError e
    = putStrLn "Pole faili lugemisõigust!"
  | otherwise
    = ioError (userError "Tekkis viga. Kas fail ikka olemas?")

püüaMuud e
  = putStrLn (ioeGetErrorString e)
```

(103)

Kasutatud operaatori `ioeGetErrorString` väärtus on funktsioon, mis võtab argumendiks erindi ja annab tulemuseks erindit iseloomustava stringi. Töötab ka tavaline `show`, tema väärtus võib olla mõnevõrra teistsugune.

Operaatorite `userError` ja `ioError` järjestrakendamise jaoks on olemas ka eraldi operaator

`fail`. Koodis (103) võiks niisiis avaldise

```
ioError (userError "Tekkis viga. Kas fail ikka olemas?")
```

asemel kirjutada lihtsamalt

```
fail "Tekkis viga. Kas fail ikka olemas?".
```

Ülesandeid

289. Küsida interaktiivses keskkonnas muutujate `catch`, `ioError`, `userError` tüübid ja saada neist aru.

290. 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.

291. Kirjutada ülesande 290 lahenduseks olev programm ümber nii, et tekkinud erindite kirjeldamisel mainitaks faili nime (nt kujul “Failil “k.txt” puudub lugemisõigus!”).

292. Kirjutada programm, mis küsib käivitudes kasutajalt kaks failinime ja kirjutab esimese faili need read, milles esineb täht `a`, teise faili. Kui faili lugemine või kirjutamine ebaõnnestus, lõpetab programm samuti kohe töö vastavasisulise teatega ekraanil. Püüda tekkinud viga võimalikult täpselt diagnoosida.

Kõrgemat järku funktsioonide defineerimine

Kõrgemat järku funktsiooni definitsioon ei erine väliselt millegi poolest tavaliste funktsioonide definitsioonidest. Mingeid erilisi süntaktilisi konstruktsioone vaja ei lähe. Tuleb teada, et funktsioonitüüpi argumendi näidiseks kõlbab reaalselt ainult kas muutujanäidis või jokker. Konstruktoritega näidised on keelatud, ehknäidis ja laisk näidis aga ei oma sellisel juhul mõtet.

Deklaratsioon

```
kõrgem
  :: (Num a)
  => (a -> b) -> b                                    (104)
kõrgem f
  = f 1
```

defineerib muutuja `kõrgem` väärtuseks kõrgemat järku funktsiooni, mis võtab argumendiks arvulise argumenditüübiga funktsiooni ja annab väärtuseks selle funktsiooni väärtuse kohal 1. Funktsioonitüüpi argumendi näidiseks on muutuja `f`. Argumentfunktsiooni väärtusetüüp ja ühtlasi defineeritava funktsiooni väärtusetüüp võib olla mis iganes.

Näiteks avaldise `kõrgem log` väärtus on arv 0, sest $\log 1 = 0$. Samuti on `kõrgem (5 +)` ja `kõrgem (5 -)` korrektsed avaldised: esimese väärtus on 6, teise väärtus 4. Korrektne avaldis on ka `kõrgem properFraction`, mille väärtus on paar $(1\,,0)$, sest `properFraction 1` on korrektne ja väärtuseks on arvu 1 täisosa ja murdosa paarina ehk $(1\,,0)$. Seevastu `kõrgem fst` on tüübivigane, sest `fst` väärtuseks oleva funktsiooni argumenditüüp pole arvuline, see on paaritüüp.

Küll aga saab operaatorit `kõrgem` rakendada argumentidele, mille väärtuseks on mingi *curried*-kujul funktsioon f, kui vaid f võtab esimesena argumendiks arvulist tüüpi objekti. Sel juhul on `kõrgem` rakendamisel saadud avaldise väärtuseks funktsioon, mis jääb üle f rakendamisel arvule 1. Näiteks aritmeetikatehted kõik sobivad. Nii näiteks on korrektne avaldis `kõrgem (-)`, mille väärtus on sama mis avaldisel `(-) 1` ehk funktsioon, mis lahutab oma argumendi arvust 1.

Argumentide arv muutuja definitsioonis ei määra, milline peab olema argumentide arv tema kasutuses. Argumentide arvule seavad piiranguid ainult tüübid. Kuna `kõrgem (-)` väärtus on funktsioon, siis saame avaldisele `kõrgem (-)` tüübikorrektselt veel ühe argumendi anda, kuigi definitsioonis (104) on muutujal `kõrgem` näha ainult üks argument. Näiteks `kõrgem (-) 5` on igati korrektne avaldis, mille väärtus on -4.

Ülesandeid

293. Otsida Hugsi teegist mooduli Prelude algtekstist üles operaatorite `$`, `flip`, `curry`, `uncurry`, `.` definitsioonid ja saada neist aru.

294. Defineerida oma moodulis muutuja `?` väärtuseks postfiksne funktsioonirakendamine (st ta peab võtma funktsiooni oma argumendi järel (vastandina operaatorile `$`)).

295. Olgu muutuja `kõrgem` antud definitsiooniga (104). Milliseid avaldistest

    ```
    (: [88]), take, takeWhile, foldr, foldr (+)
    ```

    saab tüübikorrektselt anda `kõrgem` argumendiks? Mis on `kõrgem` rakendamisel saadavate avaldiste väärtus neil argumentidel, kui see on tüübikorrektne?

Koodiga (93) defineerisime muutuja `aste` väärtuseks arvude astendamise täisarvuga. See kasutas asjaolu, et astendamine esitub ühe ja sama arvu paljukordse korrutamise kaudu.

Ühe fikseeritud objektiga sama operatsiooni paljukordse sooritamise näiteid on matemaatikas palju — olgu siin ühe silmatorkavaimana mainitud korrutamine, mis esitub samal moel liitmise kaudu —, mistõttu on mõttekas see algoritmiskelett välja abstraheerida.

Defineerime muutuja `kõrgAste` väärtuseks kõrgemat järku funktsiooni, mis suudab leida mistahes etteantud üht tüüpi argumentidega binaarse operatsiooni naturaalarv korda sooritamise tulemusi. Lisaks astme alusele ja astendajale võtab ta argumendiks binaarse operatsiooni, millega

kokkuarvutamine toimub, ja objekti, mis tuleb vastuseks anda 0-ga astendamise puhul. Jätame vaatluse alt välja astendamise negatiivse täisarvuga, mille jaoks meil oleks vaja veel ka pöördoperatsiooni.

Valdavalt koodi (93) järgi tehes saame definitsiooniks

```
kõrgAste
  :: (Integral b)
  => (a -> a -> a) -> a -> a -> b -> a
kõrgAste (*) e a n
  = case compare n 0 of
      GT                                                    ,   (105)
        -> a * kõrgAste (*) e a (n - 1)
      EQ
        -> e
      _
        -> error "kõrgAste: negatiivne astendaja"
```

kus `kõrgAste` esimene argument `(*)` tähistab operatsiooni ja `e` arvuga 0 astendamise tulemust. Argumendid `a` ja `n` on, nagu ka definitsioonis (93), astme alus ja astendaja.

Pangem tähele, et muutuja `*` ei ole definitsioonis (105) oma tavatähendusega, sest ta esineb formaalses parameetris ja on seetõttu siin definitsioonis seotud lokaalselt. Tema väärtuseks tuleb milline iganes operatsioon, mis `kõrgAste` väljakutsel talle argumendiks antakse.

Tavalise täisarvulisse astmesse tõstmise operaatori saab nüüd defineerida koodiga

```
aste
  :: (Num a, Integral b)
  => a -> b -> a                                            (106)
aste a n
  = kõrgAste (*) 1 a n
```

erijuhuna kõrgemat järku astendamisest. Argumentoperatsiooniks oleme sellega fikseerinud korrutamise ja 0-ga astendamise tulemuseks arvu 1. Definitsioonis (106) tähendab `*` tavalist korrutamist, sest ta pole lokaalselt seotud muus tähenduses. See tähendab, et `kõrgAste` väljakutsel definitsioonist (106) saab ka definitsiooni (105) lokaalne muutuja `*` väärtuseks tavalise korrutamise.

Soovi korral võime defineerida teisi analoogilisi operaatoreid, andes `kõrgAste` argumentideks midagi muud. Näiteks saab täisarvuga korrutamise defineerida korduva liitmisoperatsiooni kaudu koodiga

```
korrutis
  :: (Num a, Integral b)
  => a -> b -> a                                            (107)
korrutis a n
  = kõrgAste (+) 0 a n
```

ja superastendamise korduva tavaastendamise kaudu koodiga

```
super
  :: (Floating a, Integral b)
  => a -> b -> a                .                    (108)
super a n
  = kõrgAste (**) 1 a n
```

Kui `kõrgAste` on kutsutud välja definitsioonist (107), siis definitsiooni (105) lokaalse muutuja `*` väärtus on liitmine, kui aga definitsioonist (108), siis ujukomaarvude astendamine.

Koodi (105) korrektseks kasutamiseks on vaja `e` väärtuseks valida `*` väärtuseks oleva operatsiooni parempoolne ühik. Näiteks astendamise esitamisel korduva korrutamise kaudu oli selleks korrutamise ühik ehk 1, korrutamise esitamisel korduva liitmise kaudu aga liitmise ühik ehk 0. Superastendamise definitsioonis oli selleks jälle 1, sest 1 on astendamise parempoolne ühik (iga arv astmes 1 on arv ise). Kui selle printsiibi eest mitte hoolt kanda, hakkab `e` väärtus tulemust mõjutama.

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 järjestrakendamine ehk saba saba võtmine. Et tegu on funktsiooniga, saame tõrgeteta lisada argumendi, näiteks

```
kõrgAste (.) id tail 2 [1, 2, 3, 4, 5]
```

väärtus on $3 : 4 : 5 : []$.

Astendamisest oli meil olemas ka “jaga ja valitse” metoodikaga realisatsioon (94). See on üldistatav samamoodi nagu kood (93). Koodi (94) ümbertegemine annab uue definitsiooni

```
kõrgAste (*) e a n
  = case compare n 0 of
      GT
        -> let
             (q , r)
               = divMod n 2
             z = kõrgAste (*) e a q              .       (109)
           in
           if r == 0 then z * z else a * z * z
      EQ
        -> e

      _
        -> error "kõrgAste: negatiivne astendaja"
```

Muutuja `kõrgAste` sellise definitsiooni korral töötavad koodiga (106) defineeritud operaator `aste` ja teised sarnaselt defineeritud operaatorid palju kiiremini. Võib rääkida logaritmilisest keerukusest, kuid siis tuleb täpsustada, et keerukus tähendab operaatori `*` rakendamiste arvu. See ei lange asümptootiliselt kokku ajalise keerukusega, sest üldjuhul mida suurem on astendaja, seda suuremaks lähevad arvutuse käigus argumendid ja seda aeganõudvam on operatsiooni ühekordne sooritamine.

Äärmiselt oluline on arvestada, et "jaga ja valitse" tehnikaga on astet võimalik arvutada vaid tänu korrutamisoperatsiooni assotsiatiivsusele, sest selline astendamisalgoritm baseerub sulgude ümberpaigutamisel. Seega annab taolise definitsiooniga nagu (109) arvutamine korrektseid tulemusi vaid assotsiatiivsete argumentoperatsioonide korral.

Näiteks ei ole võimalik koodiga (109) defineeritud funktsiooni `kõrgAste` rakendusena saada superastendamist, sest astendamine ei ole assotsiatiivne. Kood (108) töötaks lihtsalt valesti.

Seevastu funktsioonide kompositsioon on assotsiatiivne, seega funktsioonide astmete arvutamine töötab korrektselt ka koodiga (109). Kuid funktsioonide astendamise juures ei anna astendamiseks "jaga ja valitse" tehnika 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 nüüd 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.

Ülesandeid

296. Otsida Hugsi teegist mooduli Prelude algtekstist üles muutujate `iterate`, `until`, `takeWhile`, `dropWhile`, `filter`, `map`, `zipWith`, `foldr`, `foldl`, `scanl` definitsioonid ja saada neist aru.

297. Kasutades koodiga (105) või (109) defineeritud muutujat `kõrgAste`, kirjutada interaktiivse interpretaatori käsurealt avaldis, mille väärtus saadakse siinuse 100-kordsel rakendamisel arvule 1.

298. Defineerida muutuja `kõrgFact` 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 `kõrgFact (^) 1 n` väärtus, kui `n` väärtus on naturaalrv n, peab olema $n^{(n-1)^{\cdots^1}}$. Väljendada `kõrgFact` kaudu faktoriaalifunktsioon ning binoomkordajaid $\binom{n}{2}$ leidev funktsioon.

299. 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 $m \oplus n$.

300. 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.

301. Defineerida otserekursiooniga muutuja `loeDropWhile` väärtuseks funktsioon, mis võtab järjest argumendiks predikaadi p ja listi l ning annab välja paari, mille esimene komponent on list, milles on l elemendid alates esimesest sellisest, mis tingimust p ei rahulda (tühi list, kui sellist ei leidu), ja teine komponent näitab l algusest äravisatud elementide arvu.

302. Defineerida muutuja `esimKuiOn` väärtuseks *curried*-kujul funktsioon, mis võtab argumentideks predikaadi p ja listi l: kui listis l leidub element, mis rahuldab predikaati p, siis annab tulemuseks väärtuse Just x, kus x on esimene p-d rahuldav element l-s; vastasel korral kui list on lõplik, siis annab tulemuseks Nothing.

303. Anda standardteegis defineeritud muutujale `scanl` oma moodulis uus samaväärne definitsioon rekursiivse listiga. Testida mittekommutatiivsete operatsioonidega.

304. Defineerida funktsioon `pikkZipWith`, mis töötab nagu `zipWith`, aga ei viska ülejäävat listisaba ära, vaid jätab selle tulemuslisti lõppu.

305. Defineerida muutuja `värskenda` väärtuseks *curried*-kujuline funktsioon, mis võtab argumentideks funktsiooni f, listi l ja arvu i ning annab tulemuseks listi, mille saab listist l tema i-ndale komponendile (lugemine alates 0-st) funktsiooni f rakendamisel.

306. Defineerida muutuja `mapUntil` väärtuseks *curried*-kujuline 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.

307. Defineerida muutuja `lõiguPoolitamine` väärtuseks kõrgemat järku funktsioon, mis võtab argumendiks funktsiooni ja leiab tema nullkoha lõigu poolitamise meetodil. Kood peab töötama juhul, kui argumentfunktsioon on kasvav ja tal on nullkoht.

308. Kasutades ülesandes 307 defineeritud muutujat `lõiguPoolitamine`,

    - lahendada uuesti ülesanne 244;
    - defineerida muutuja `poolringiPoolitaja` väärtuseks funktsioon, mis võtab argumendiks ujukomaarvu a: kui a on 0 ja 1 vahel, siis leiab nurgasuuruse α, mille korral poolringi segment, mis ühelt poolt piirneb diameetri ja teiselt poolt kõõluga, mille otspunkt on diameetri otspunktist kaart pidi kaugusel α, moodustab pindalalt a osa kogu poolringist.

309. Defineerida muutuja `kahekaupaMap` väärtuseks kõrgemat järku funktsioon, mis võtab argumentideks *curried*-kujul funktsiooni $\oplus$ ja listi l ning annab tulemuseks listi, mille saab, võttes listi l elemendid kahekaupa kokku ja rakendades igale paarile operatsiooni $\oplus$ (kui l elementide arv on paaritu, siis viimane element jääb niisama tulemuslisti lõppu).

310. Kasutades ülesande 309 lahendust, defineerida muutuja `puuFold` väärtuseks kõrgemat järku funktsioon, mis võtab argumendiks operatsiooni $\oplus$ ja listi l ning arvutab listi l elemendid operatsiooniga $\oplus$ kokku, asetades sulud selliselt, et avaldise struktuur oleks puulaadne. Täpsemalt, avaldise struktuur peab olema kahendpuu, kus igal hargneval tipul on olemas kaks alluvat ning iga sellise tipu vasak alluv on balansseeritud ja sisaldab vähemalt samapalju lehti kui parem alluv. Näiteks kui listi elemendid on a_1, a_2, a_3, a_4, on tulemuseks $(a_1 \oplus a_2) \oplus (a_3 \oplus a_4)$; kui listi elemendid on a_1, a_2, a_3, a_4, a_5, on tulemuseks $((a_1 \oplus a_2) \oplus (a_3 \oplus a_4)) \oplus a_5$; kui listi elemendid on $a_1, a_2, a_3, a_4, a_5, a_6$, on tulemuseks $((a_1 \oplus a_2) \oplus (a_3 \oplus a_4)) \oplus (a_5 \oplus a_6)$; kui listi elemendid on $a_1, a_2, a_3, a_4, a_5, a_6, a_7$, on tulemuseks $((a_1 \oplus a_2) \oplus (a_3 \oplus a_4)) \oplus ((a_5 \oplus a_6) \oplus a_7)$.

311. Realiseerida järjestamine põimimismeetodil ülesandes 310 defineeritud muutuja `puuFold` abil.

312. Defineerida muutuja `kõrgJärjestatud` väärtuseks kõrgemat järku funktsioon, mis võtab argumendiks binaarse võrdlusoperatsiooni $\leq$ ja annab välja funktsiooni, mis võtab argumendiks listi ja annab välja True või False vastavalt sellele, kas list on lõplik ja $\leq$ mõttes mittekahanevalt järjestatud või leidub listis koht, kus element on talle järgnevast $\leq$ mõttes suurem.

313. Realiseerida kõrgemat järku järjestamine, mis võtab võrdlusoperatsiooni argumendiks, kiirmeetodil ja põimimismeetodil.

Kõrgemat järku funktsioonide kasutamisel on tihti kõige raskemaks osaks äratabamine, et üks või teine operatsioon on erijuht teatavast lihtsast kõrgemat järku funktsioonist. Selle iseloomustamiseks näitame nüüd, kuidas on võimalik kõrgemat järku astendamist kasutada Fibonacci arvude arvutamiseks. Täpsemalt, eesmärgiks on defineerida muutuja `fib` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja annab tulemuseks Fibonacci jada n-nda liikme (sama tegid ka mitmed varasemad definitsioonid nagu (52), (53), (83) ja (87)).

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

$$\mathscr{F}^n = \begin{pmatrix} F_{n-1} & F_n \\ F_n & F_{n+1} \end{pmatrix}. \tag{110}$$

Tõepoolest, see ilmselt kehtib $n = 1$ korral ning kui ta kehtib mingi i jaoks, siis

$$\mathscr{F}^{i+1} = \mathscr{F} \cdot \mathscr{F}^n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} \cdot \begin{pmatrix} F_{i-1} & F_i \\ F_i & F_{i+1} \end{pmatrix} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i-1} + F_i & F_i + F_{i+1} \end{pmatrix} = \begin{pmatrix} F_i & F_{i+1} \\ F_{i+1} & F_{i+2} \end{pmatrix}.$$

Seega tuleks realiseerida 2×2-maatriksite korrutamine. Selle töö lihtsustamiseks tasub teha veel üks tähelepanek. Kirjutades võrduses (110) F_{n-1} kohale samaväärselt $F_{n+1} - F_n$, me näeme, et $\mathscr{F}$ kõik astmed on kujul

$$\begin{pmatrix} y - x & x \\ x & y \end{pmatrix}, \tag{111}$$

kus x, y on mingid täisarvud. Järelikult on meisse puutuvate maatriksite ülemine rida alumise reaga üheselt määratud ja me võime kõik vajalikud tegevused ära 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 (111), 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 (110) 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)
```
(112)

Kuna maatriksite korrutamine on assotsiatiivne ja meie operatsioon paaridel peegeldab maatriksite korrutamist üksühele, on ka see operatsioon paaridel assotsiatiivne ja on võimalik kasutada `kõrgAste` definitsiooni (109). Seega viimatiesitatud kood arvutab Fibonacci arve logaritmilise keerukusega järjekorranumbri suhtes ja ületab oma efektiivsuselt kõik varasemad, isegi need, mis lasid Fibonacci arve listist otsida, sest listist otsimine on lineaarse keerukusega.

Kui kasutada siiski `kõrgAste` lineaarse keerukusega definitsiooni (105), siis arvutuse käigus läbitakse täpselt samad vahetulemused nagu näiteks definitsioonide (53) ja (98) korral, mis itereerisid operatsiooni, mis seab igale paarile (a, b) vastavusse 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 $(b, a + b)$.

Väärtuste maailma laiendamine

Algebralised tüübid

Seni oleme oma tegemisi teinud maailmas, mis on piiratud Haskelli eeldefineeritud väärtuste ruumiga. Meie definitsioonid ei loonud uusi väärtusi, vaid omistasid juba olemasolevaid väärtusi uutele muutujatele.

Haskell võimaldab aga ka väärtuste maailma laiendada. Programmeerija saab võtta kasutusele täiesti uusi väärtusi nii andmete kui ka tüüpide tasemel. Andmete ja tüüpide maailma rikastamine käib seejuures alati koos. Pole võimalik näiteks uute andmete lisamine juba olemasolevasse tüüpi.

Uute väärtuste sissetoomine tähendab ühtlasi uute konstruktorite defineerimist: selleks, et uusi väärtusi vanadest eristada, tuleb neid koostada uute, endistest erinevate konstruktorite abil.

Tüüpe, mida saab konstrueerida Haskellis, nimetatakse algebralisteks (ingl *algebraic*).

Vabadus uusi väärtusi konstrueerida võimaldab objekte, 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.

Tüüpide defineerimine

Enamasti sobib uute tüüpide ja neisse kuuluvate väärtuste sissetoomiseks deklaratsioon kujul

```
data t
  = a1  |  ...  |  an .
```
(113)

Deklaratsioon (113) peab olema moodulis välistasemel, ta ei saa olla lokaalne. Lisaks peab kehtima tingimus $n > 0$.

Vaatleme algul lihtsamat juhtu, kus defineeritakse korraga ainult üks tüüp. Siis kujutab $\mathfrak{t}$ endast uue tüübi nime. Kuna süntaktiliselt on tegemist tüübikonstruktoriga, mitte muutujaga, peab nimi algama suurtähega.

Püstkriipsudega eraldatud osad $\mathfrak{a}_1, \ldots, \mathfrak{a}_n$ on **alternatiivid**. Iga alternatiiv esindab uude tüüpi kuuluvate andmete või andmestruktuuride üht võimalikku kuju. Koos kirjeldavad nad kõik sellesse tüüpi kuuluvad normaalsed objektid; neile lisandub veel ebanormaalne väärtus $\perp$.

Täpsemalt, iga andmete kuju eristab teistest uus unikaalne andmekonstruktor, mille rakendamisel just need andmed saadakse. Konstruktori argumendid on sisuliselt andmestruktuuri **väljad** (ingl *field*), mille konstruktor lihtsalt üheks andmestruktuuriks kokku seob, analoogiliselt näiteks sellega, kuidas paarikonstruktor oma kaks argumenti üheks paariks kokku seob. Alternatiiv deklaratsioonis (113) 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 tüübisignatuure anda ei saa.

Andmekonstruktorite nimed peavad olema reeglipärased. Nende rakendus võib olla *curried*-kujuline ja seejuures vastavalt soovile prefiksne või infiksne. Andmekonstruktori argumentide arv võib olla ka 0; see tähendab, et ühtegi rakendamist ei toimu, konstruktor ise esindab väärtust uuest tüübist. Alternatiiv koosneb siis paljast andmekonstruktorist.

Kõige lihtsamal juhul pole ühelgi konstruktoril argumente. Siis on defineeritav tüüp loetelutüüp.

Näiteks deklaratsioon

```
data Kuu
  = Jaanuar
  | Veebruar
  | Märts
  | Aprill
  | Mai
  | Juuni
  | Juuli
  | August
  | September
  | Oktoober
  | November
  | Detsember
```
(114)

defineerib uue loetelutüübi Kuu, milles on 12 väärtust. Need väärtused on seotud uute konstruktoritega `Jaanuar`, `Veebruar`, `Märts`, `Aprill`, `Mai`, `Juuni`, `Juuli`, `August`, `September`, `Oktoober`, `November`, `Detsember`.

Kuutüüpi objekte saab kasutada kalendrikuudega seotud arvutuste programmeerimiseks, kui on soov eristada kalendrikuid muudest väärtustest ja mitte kasutada kuude tähistamiseks näiteks

täisarve 1 kuni 12. Inimestena võtame objekte tüübist Kuu kui kalendrikuid; masina jaoks on tegu lihtsalt mingite andmetega, mis eristuvad kõigisse teistesse tüüpidesse kuuluvatest andmetest oma unikaalsete nimede poolest.

Eraldi kalendrikuutüübi 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 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 väärtusi LT, EQ, GT loetelutüübist Ordering, kuigi saaks kasutada ka arve, samuti miks tõeväärtustel on oma tüüp jne.

Deklaratsiooni (114) olemasolul on kuutüüp oma konstruktoritega põhimõtteliselt kohe kasutatav. Kuid lastes interaktiivses keskkonnas väärtustada näiteks konstruktori `Jaanuar`, saame oodatava vastuse `Jaanuar` asemel hoopis tüübivea, mis tuleneb sellest, et uus tüüp ei kuulu klassi Show. Kui meil pole erilisi soove selles osas, kuidas kuutüüpi objekte tuleks stringiks teisendada, võib kasutada automaatset klassikuuluvuse tuletust. Selleks piisab definitsiooni (114) lõppu lisada rida

```
deriving (Show).                                    (115)
```

Selle tulemusel saab kuutüüp klassi Show esindajaks, kusjuures kuutüüpi väärtuste stringikujud tuletatakse otse konstruktorite nimedest. Andes nüüd interaktiivse keskkonna käsurealt `Jaanuar`, saame ka vastuseks `Jaanuar`.

Edasi võiksime kirjutada näiteks koodi

```
suvekuud
  :: [Kuu]
suvekuud                    ,
  = [Juuni, Juuli, August]
```

mis defineerib muutuja `suvekuud` väärtuseks suvekuude listi. Kui aga on tarvis funktsiooni, mis võtab kalendrikuid argumendiks, siis võib kasutada kuukonstruktoreid näidistena. Näiteks 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
```

(116)

Definitsioon (116) 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. Sellise ideega 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. Jällegi aitab tüübiklassikuuluvuse automaatne tuletus: kui lisada täiendi (115) klasside loendisse ka `Eq`, defineerub automaatselt kuutüübil võrduspredikaat, mis loeb iga väärtuse võrdseks parajasti iseendaga.

Samamoodi saab tekitada uute tüüpide kuuluvust ka klassi Ord. Kuutüübi puhul tekib klassi Ord kuuluvuse automaatsel tuletamisel võrdlusoperatsioon, mille järgi aastas eespool paiknevad kuud on tagapool paiknevatest väiksemad. Süsteem tuletab niisuguse võrdluse selle põhjal, millises järjestuses konstruktorid definitsioonis (114) on esitatud. See näitab, et tüübiklassikuuluvuse automaatse tuletamise puhul on alternatiivide järjekord definitsioonis oluline.

Loetelutüüpide puhul saab automaatselt tuletada ka kuuluvust klassi Enum. See annab võima-

luse loetelutüüpi objektide puhul opereerida nende järjekorranumbritega. Esimene konstruktor saab järjekorranumbri 0, järgmine 1 jne, nii et kuutüübi korral ollakse ühe võrra nihkes võrreldes traditsioonilise numeratsiooniga.

Edaspidi on oluline kuutüübi kuuluvus kõigisse mainitud klassidesse, nii et eeldame, et definitsiooni (114) on täiendatud reaga

```
  deriving (Show, Eq, Ord, Enum).
```

Tänu kuulumisele klassi Enum saab muutujale `päevi` anda varasemaga võrreldes üsna lühikese definitsiooni

```
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
```
, (117)

muutujale `suvekuud` aga saab anda uue definitsiooni

```
suvekuud
  = [Juuni .. August].
```

Klassi Enum kuuluvate tüüpide puhul on defineeritud ka muutujad `succ` ja `pred`, mis tähendavad vastavalt järgmise ja eelmise väärtuse võtmist. Neist esimese abil saame kirjeldada funktsiooni, mis leiab etteantud kuule järgneva kuu, definitsiooniga

```
järgmKuu
  :: (Integer , Kuu) -> (Integer , Kuu)
järgmKuu (y , m)
  = case m of
      Detsember
        -> (succ y , Jaanuar)
      _
        -> (y , succ m)
```
. (118)

Ülesandeid

314. Otsida Hugsi teegist üles tüüpide Bool ja Ordering definitsioonid ja saada neist aru.

315. Kirjutada definitsioon (114) oma moodulisse ja viia kuutüüp klassi Enum. Defineerida muutuja `kõikKuud` väärtuseks list, milles on aastas esinemise järjekorras kõik kalendrikuud.

316. Defineerid muutuja `eelmKuu` väärtuseks funktsioon, mis leiab etteantud kuule eelneva kuu. Tüüp peab olema sama mis koodiga (118) defineeritud muutujal `järgmKuu`.

317. Defineerida loetelutüüp `Nädalapäev`, millesse kuuluvad väärtused vastavad nädalapäevadele. Seejärel defineerida muutuja `tööpäev` väärtuseks predikaat, mis võtab argumendiks nädalapäeva ja annab True parajasti juhul, kui argument on tööpäev.

318. Defineerida loetelutüüp `Viker`, mille väärtused vastavad värvidele värviringis. Defineerida muutuja `järgm` väärtuseks funktsioon, mis võtab argumendiks värvi ja annab tulemuseks ringis järgmise värvi. Analoogselt defineerida ka muutuja `eelm` ringis eelmise värvi leidmiseks.

Teine lihtne juht deklaratsioonist kujul (113) on selline, kus on ainult üks alternatiiv. Siis kõik normaalsed väärtused uuest tüübist konstrueeritakse sama konstruktoriga. Selliste tüüpide hulka kuuluvad senituntutest näiteks paaritüüp, kus kõik normaalsed väärtused on konstrueeritud paarikonstruktoriga.

Näiteks võib lisaks kuutüübile anda kuupäevatüübi deklaratsiooniga

```
data Daatum
  = Daatum Integer Kuu Int.
  deriving (Show, Eq, Ord)
```
(119)

Kuupäevatüübi nimeks on valitud `Daatum`, see nähtub deklaratsiooni (119) võrduse vasakust poolest. Parem pool ütleb, et iga väärtus sellest tüübist konstrueeritakse konstruktoriga `Daatum`, mis võtab kolm argumenti: täisarvu, kalendrikuu ja veel ühe täisarvu. Esimese täisarvu mõte on mängida aastaarvu rolli, viimase täisarvu mõte on mängida päeva numbri rolli.

Niisiis on nimi `Daatum` deklaratsiooni (119) erinevates pooltes kasutatud erinevas tähenduses: vasakul on ta tüüp ehk 0 argumendiga tüübikonstruktor, paremal aga 3 argumendiga andmekonstruktor. Andmekonstruktor `Daatum`, lähtuvalt tema argumentide tüüpidest ja sellest, et tulemuseks on `Daatum`-tüüpi objekt, on tüüpi `Integer -> Kuu -> Int -> Daatum` (selles tüübiavaldises tähendab `Daatum` tüüpi). See, et tüübi nimi ja andmekonstruktori nimi langevad kokku, ei sega kuidagi, sest konteksti järgi on alati selge, kas konkreetne nime `Daatum` esinemine tähendab tüüpi või annet. Tavaliselt valitaksegi ühe alternatiiviga tüüpide puhul andme- ja tüübikonstruktorile sama nimi, aga muidugi ei ole see kohustuslik.

Näiteks oleks Tartu rahu sõlmimise päev `Daatum`-tüüpi avaldisena üles kirjutatav kujul `Daatum 1920 Veebruar 2`.

Kasutades ka varem defineeritud operaatoreid `päevi` ja `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
```
(120)

Jällegi on omadefineeritud konstruktorit kasutatud argumendinäidises. See näidis sobitub iga normaalse kuupäevatüüpi objektiga 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
  = A Integer
data Päev
  = P Int
```
(121)

ja kasutades definitsioonis (119) neid. Tüüpidesse `Aasta` ja `Päev` kuuluvad objektid 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 koodi (121) 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 (119) antud kuupäevatüüpi edasi. Viidatud puuduse kompenseerimiseks toome sisse predikaadi, mis kontrollib, kas etteantud kuupäevatüüpi objekt väljendab reaalset kuupäeva. Seda teeb näiteks kood

```
olemas
  :: Daatum -> Bool
olemas (Daatum y m d)          .
  = 0 < d && d <= päevi (y , m)
```

Päeva legaalsust tuleks kontrollida enne selliste operaatorite nagu `järgmPäev` väljakutset, muidu võime saada ootamatuid tulemusi.

Kuna deklaratsiooniga (119) tuletasime ühtlasi kuupäevatüübi kuuluvuse klassidesse Eq ja Ord, saab ka kuupäevade võrdust kontrollida ja neid omavahel võrrelda. Võrduse automaatsel tuletamisel loetakse võrdseteks ainult eristamatud objektid. Selline võrdus kuupäevatüübil vastab päevade reaalsele võrdusele, sest üht ja sama päeva esitab ainult üks kuupäevatüüpi objekt.

Automaatselt tuletatav 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. Kui tegelda ainult legaalsete kuupäevadega, siis vastab see järjestus ilmselt päevade ajalisele järgnevusele. Siin mängib otsustavat rolli asjaolu, et kuupäeva esituses antakse kõigepealt aastanumber, siis kuu ja lõpuks päevanumber.

Pangem tähele, et kuupäevatüübi viimine automaatselt klassidesse Show, Eq ja Ord on võimalik vaid tänu sellele, et kõik andmekonstruktori `Daatum` argumenditüübid (Integer, Kuu ja Int) kuuluvad samuti klassidesse Show, Eq ja Ord, sest nii stringiksteisendamisel kui ka võrdlemisel tuleb alamülesandena teisendada stringiks või võrrelda välju. Kui väljatüübid neisse klassidesse ei kuuluks, tekitaks automaatse tuletuse nõudmine deklaratsioonis (119) tüübivea.

Ülesandeid

319. Defineerida muutuja `iseseisvus1` väärtuseks funktsioon, mis võtab argumendiks kuupäeva ja kontrollib, kas see kuulub Eesti esimesse iseseisvusaega.

320. 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.

321. Defineerida muutuja `vaheAastates` väärtuseks funktsioon, mis võtab argumendiks objektid d_1, d_2 tüübist Daatum ja annab väärtuseks arvu, mitu täisaastat on kuupäeval d_2 möödas kuupäevast d_1.

322. Defineerida muutuja `vahePäevades` väärtuseks funktsioon, mis võtab argumendiks objektid d_1, d_2 tüübist Daatum ja annab väärtuseks päevade arvu kuupäevade d_1 ja d_2 vahel.

323. Defineerida aasta- ja päevanumbritüübid koodiga (121) ja teha päevatüübi definitsioon (119) ümber nii, et uued tüübid oleksid sobivalt kasutatud. Teha ümber definitsioonid (117) ja (118), nii et nad sobivalt kasutavad aastanumbritüüpi. Vajadusel teha ümber ka definitsioon (120), nii et ta töötaks muutunud päevatüübil.

324. Kasutades ülesandes 317 defineeritud nädalapäevatüüpi, defineerida muutuja `nädalapäev` väärtuseks funktsioon, mis võtab argumendiks kuupäeva ja annab tulemuseks nädalapäeva, millele see kuupäev satub.

325. Nädalakalendris jaotatakse päevad mitte kuudesse, vaid nädalatesse. Päevad identifitseeritakse aastaarvu, nädala numbri ja nädalapäeva kaudu. Nädala kuuluvuse aastasse määrab tema neljapäeva kuuluvus Gregooriuse kalendri sama numbriga aastasse. Ühe aasta piires tähistatakse nädalaid järjestikuste täisarvudega alates 1-st.

    Kasutades ülesande (324) lahendust ja ülesandes 317 defineeritud nädalapäevatüüpi, defineerida nädalakalendripäevatüüp. Defineerida muutuja `olemas` väärtuseks predikaat, mis võtab argumendiks sellist nädalakalendripäevatüüpi objekti ja annab välja tõeväärtuse vastavalt sellele, kas selline päev on nädalakalendris olemas või mitte.

Näitena tüübidefinitsioonist, mis sisaldab mitu alternatiivi, mille konstruktoritel on argumendid, anname deklaratsiooni

```
data Kalendripäev
  = Gregooriuse Daatum
  | Juuliuse Daatum    .
  deriving (Show)
```
(122)

Väärtused nii defineeritud tüübis on kuupäevad konkreetses kalendris — kas gregooriuse või juuliuse ehk, nagu öeldakse, vastavalt uues või vanas kalendris. Sellist tüüpi võib vaja minna juhul, kui soovitakse päevi talletada selle kalendri järgi, mis sel päeval kehtis. Definitsioon (122) 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).
```

Deklaratsiooni (113) paremas pooles võib esineda ka keerulisemaid 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]
```
(123)

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 objektid, kus kaht kuupäeva võib mõista kui alumist ja ülemist ajalist tõket. Kolmandaks on konstruktoriga `Valik` ühest kuupäevade listi tüüpi väljast ehitatud objektid, kus listis on kõik võimalikud kuupäevad ükshaaval üles loetud. Sellist laadi väärtust esitab näiteks avaldis `Valik [Daatum 1918 Veebruar 24, Daatum 1991 August 20]`.

Kirjeldame kasutusnäitena predikaadi, mis võtab argumendiks osalise info ehk `EbaDaatum`-tüüpi objekti p ja kuupäeva d tüübist `Daatum` ning kontrollib, kas osaline info p lubab kuupäeva d. Eeldame, et olematuid kuupäevi ei kasutata. Sobib kood

```
lubab
  :: EbaDaatum -> Daatum -> Bool
lubab ed d
  = case ed of
      Valik ds
        -> elem d ds
      Piirid a b
        -> a <= d && d <= b
      _
        -> True
```
. (124)

Valikuavaldise teise juhu käsitlus kasutab ära ülal selgitatud asjaolu, et päevade ajaline järgnevus vastab automaatselt tuletatud järjestusele.

Kuigi Haskell ei luba olemasolevaid tüüpe laiendada ega kasutada tüübina olemasolevate tüüpide ühendit, võimaldavad algebralised tüübid seda kunstlikult lavastada. Näiteks tüüp `EbaDaatum` on ehituse poolest kuupäevade listi tüübi ja kuupäevade paari tüübi ühend, millesse on lisatud veel üks väärtus. Tüüp `Kalendripäev` on aga kuupäevatüübi ühend oma koopiaga.

Ülesandeid

326. Defineerida muutuja `sordi` väärtuseks funktsioon, mis võtab argumendiks koodiga (122) defineeritud tüüpi objektide listi ja annab tulemuseks listi, kus argumentlistis esinevad gregooriuse kalendri kuupäevad on ees ja juuliuse kalendri kuupäevad taga. Sama kalendri kuupäevade omavaheline järjestus peab olema sama mis originaallistis.

327. Defineerida muutuja `valikuna` väärtuseks funktsioon, mis võtab argumendiks `EbaDaatum`-tüüpi objekti ja annab tulemuseks sama tüüpi objekti, mis kirjeldab sama päevade hulka, kuid on esitatud valikuna kuupäevade listist.

328. Defineerida uus tüüp `Üksliige`, millesse kuuluvate objektidega 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 tüüpi `String`.

Deklaratsiooniga (113) saab defineerida mitte ainult üksikuid tüüpe, vaid korraga terveid tüüpide peresid. Selleks peab selle deklaratsiooni vasakus pooles olema uus tüübikonstruktor koos argumendinäidistega. Haskell 98 lubab tüübinäidistena kasutada vaid muutujaid (seda nägime juba tüübisünonüümide defineerimisel). Parem pool on ehituselt selline nagu varem kirjeldatud, ainult et ta võib sisaldada neid muutujaid.

Vasaku poole muutujate iga väärtustuse jaoks saame ühe uue tüübi. Täpsemalt, uue tüübikonstruktori tähenduseks on *curried*-kujuline tüübifunktsioon, mis võtab samapalju argumente, kuipalju vasakus pooles on argumendinäidiseid, ja annab neil tulemuseks tüübi. Tulemustüüp koosneb objektidest, mis on kirjeldatud parema poolega samamoodi nagu varem, kusjuures lokaalsete muutujate väärtusteks on vastavad argumendid.

Suures osas on põhimõte sama nagu funktsionaalse vasaku poolega deklaratsioonide ja tüübisünonüümide deklaratsioonide puhul. Erinevalt neist deklaratsioonidest on uue tüübi definitsiooni vasak pool alati täisargumenteeritud, sest parem pool on kirjeldatud väärtuste kogumina, ta ei saa väljendada tüübifunktsiooni.

Enamasti ongi uusi tüüpe mõistlik sisse tuua perede kaupa, et võimaldada polümorfsust. Kui definitsiooni (113) vasak pool sisaldab muutujaid, on alternatiive tähistavad andmekonstruktorid automaatselt polümorfsed, nad on kasutatavad tüübimuutujate kõigi väärtuste korral. Aga ka andmemuutujaid saab defineerida polümorfselt, nii et nende väärtus võiks olla erinev sõltuvalt nende tüübimuutujate väärtustest. Nii konstruktorite kui ka muutujate polümorfismiga oleme varasemast tuttavad, sest samamoodi polümorfsed on näiteks listikonstruktorid `[]` ja `:`, Maybe-tüüpi väärtuste konstruktorid `Nothing` ja `Just` ning kõik muutujad, mille tüübisignatuuris esineb klassikuuluvustega piiritlemata tüübimuutuja.

Näiteks võib deklaratsioonis (123) kirjeldatud tüübiga `EbaDaatum` sarnase põhimõtte alusel koostatud tüüpi vaja minna peale kuupäevade ka teistlaadi väärtuste järgnevusvahemike ja valikute esitamiseks. Seepärast on mõistlik abstraheerida kuupäevatüüp definitsioonis (123) välja suvaliseks tüübiks. Saame deklaratsiooni

```
data Eba d
  = Teadmata
  | Piirid d d'
  | Valik [d]
```
(125)

mis defineerib ühe parameetriga tüübipere `Eba`. Konstruktori `Piirid` tüüp on `d -> d -> Eba d`, konstruktori `Valik` tüüp `[d] -> Eba d` ja konstruktori `Teadmata` tüüp lihtsalt `Eba d`.

Nüüd saame kirjutada näiteks avaldise `Piirid 0 5`, mille tüübiks võib kirjutada `Eba Integer` või üldisemalt `(Num d) => Eba d`, avaldise `Piirid (-3.5) 8.9`, mille tüübiks `Eba Double` või üldisemalt `(Floating a) => Eba d`, avaldise `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 (123) ja (125) ei saa samas moodulis esineda, sest defineerivad samad andmekonstruktorid.

Kui nüüd kirjeldame muutuja `onTeadmata` koodiga

```
onTeadmata Teadmata
  = True
onTeadmata _
  = False
```
, (126)

siis see muutuja on polümorfne, kasutatav objektide jaoks tüüpidest `Eba d` tüübimuutuja `d` suvalise väärtuse korral. Signatuuriks deklaratsiooni (126) juurde sobib

```
onTeadmata
  :: Eba d -> Bool
```
.

Muutuja `lubab` definitsioon koodist (124) on aga ilma muutusteta interpreteeritav polümorfselt, tuleb vaid tüübisignatuur ära jätta või asendada signatuuriga

```
lubab
  :: (Ord d)
  => Eba d -> d -> Bool
```
.

Ülesandeid

329. Otsida Hugsi teegist tüübiperede Maybe ja Either definitsioonid ja saada neist aru.

330. Defineerida muutuja `võimalik` väärtuseks predikaat, mis võtab argumendiks objekti deklaratsiooniga (125) defineeritud tüübist ja kontrollib, kas leidub väärtus, mis rahuldab selle objekti esitatud piiranguid.

331. Defineerida kahe parameetriga tüübipere, mille iga konkreetne tüüp sisaldab parameetritega määratud tüüpide esindajate paarid ja esimese parameetriga määratud tüüpide esindajad üksikuna. Uute konstruktorite nimed valida vastavalt tähendusele. Defineerida muutuja `paarina` väärtuseks funktsioon, mis võtab sellist tüüpi objekti argumendiks, üksikelemendid teisendab dubleerimise teel paarikujule ja paarid jätab puutumata.

332. Defineerida ülesandes 328 defineeritud tüüp ümber polümorfseks, kus konstanditüüp võiks olla suvaline ja samuti muutujatüüp.

333. Ülesandes 332 kirjutatud tüübidefinitsiooni kontekstis defineerida muutuja `isExp` väärtuseks funktsioon, mis annab True, kui argument on eksponendikujul, ja False, kui argument on astmekujul.

334. Ülesandes 332 kirjutatud tüübidefinitsiooni kontekstis defineerida muutuja `doominokuju` väärtuseks funktsioon, mis võtab argumendiks üksliikmete listi ja annab tulemuseks True või False vastavalt sellele, kas iga kaks järjestikust liiget omavad ühesugust argumenti (muutujat või konstanti), mis on eespool seisvas üksliikmes astendaja ja tagapool seisvas astendatav, või ei ole see nii.

Rekursiivselt defineeritud tüübid

Väga tihti läheb vaja andmestruktuure, mille ehitus on tsükliline selles mõttes, et tema mingi alamstruktuur on sama tüüpi kui tervik. Tüüpilised näited on mitmesugused kahendpuud ja üldse hierarhilised puud, mille alampuudeks on omakorda samalaadsed puud. Senikäsitletud andmestruktuuridest on listid sellised: iga mittetühja listi saba on omakorda list.

Sellise andmestruktuuri konstruktor peab vähemalt üheks argumendiks võtma väärtuse, mis sisaldab endas konstrueeritava struktuuriga sama tüüpi struktuuri. Nii näiteks on konstruktori `:` teine argument alati sama tüüpi list nagu tulemuslist.

Kui tahame sellise tüübi definitsiooni kirja panna kujul (113), peame paremas pooles konstruktori argumendi kohal kasutama defineeritavat tüüpi ennast. Seega on tegemist rekursiivse definitsiooniga. Arvestades funktsionaalsete keelte rekursioonilembust, on see muidugi võimalik ja nii tehaksegi. Tasub märkida, et siinkohal on rekursioon ainuvõimalik tee isegi tavapärastes imperatiivsetes keeltes.

Mis puutub listitüüpidesse, siis need on Haskelli sisse ehitatud, seega näidet nende reaalsest defineerimisest ei ole võimalik leida. Lähtudes aga deklaratsiooni (113) 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 parajasti objekti `[]` ning kõik sellised objektid, mis ehitatakse konstruktoriga `:` ühest A-tüüpi objektist ja kirjeldatavaga sama tüüpi objektist.

Defineerime sarnase, kuid juba legaalse näitena tüübid, millesse kuuluvad struktuurid on oma ehituselt samuti elementide järjendid nagu listid, kuid mille hulgas puudub tühi objekt. Alternatiivideks oleksid ühekomponendiline struktuur ja mitmest objektist koostatud järjend. Definitsiooniks sobib

```
data Joru a
  = Üks a
  | Mitu a (Joru a)
  deriving (Show)
```
. (127)

Parema poole teise alternatiivi ehitus on sama mis listitüüpide definitsioonil, kuid esimeses alternatiivis on konstruktoril üks argument. Kui paneksime esimeseks alternatiiviks palja konstruktori, saaksime listitüüpidega struktuurilt samaväärsed tüübid, esimese alternatiivi konstruktor mängiks tühja listi rolli ja konstruktor `Mitu` kooloni rolli. Definitsiooni (127) 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 komponent.

Muuhulgas kuuluvad rekursiivselt defineeritud tüüpi kohe ka lõpmatud struktuurid. Deklaratsiooni (127) olemasolul võiksime kirjutada näiteks definitsiooni

```
nullijoru
  :: Joru Int
nullijoru          '
  = Mitu 0 nullijoru
```

mis annab muutuja `nullijoru` väärtuseks lõpmatu `Joru`-tüüpi struktuuri, kus kõik komponendid on nullid.

Kui tüüp sisaldab väärtusi, mille pärisosana esinevad sama tüüpi väärtused, tuleb sellel tüübil töötavate funktsioonide defineerimisel loomuldasa kasutada rekursiooni. Nii on see listide puhul ja samuti äsjadefineeritud tüübi puhul. Näiteks funktsiooni, mis leiab kogu jorustruktuuri komponentide arvu, defineerib rekursiivne kood

```
joruPikkus
  :: Joru a -> Int
joruPikkus (Mitu _ xj)
  = 1 + joruPikkus xj '
joruPikkus _
  = 1
```

komponentide summa aga defineerib kood

```
joruSumma
  :: (Num a)
  => Joru a -> a
joruSumma (Mitu x xj).
  = x + joruSumma xj
joruSumma (Üks x)
  = x
```

Vaatame siin veel näitena kolme sagedasti vajaminevat puutüüpi, millest kaks on kahendpuud, üks aga paljuhargnev puu.

Kahendpuu (ingl *binary tree*) on puu, mis kas on tühi (ei sisalda ühtki tippu) või sisaldab juurtipu ning tema **vasaku** (ingl *left*) ja **parema** (ingl *right*) alampuu, mis kumbki on omakorda kahendpuu. See määratlus näitab vaid struktuuri ilma andmeteta; andmete hoidmiseks puustruktuuris on mitu mõistlikku võimalust, nt andmed kõigis tippudes *vs* andmed ainult lehtedes.

Kui andmeid hoitakse puu kõigis tippudes, siis sobib kahendpuutüüp defineerida koodiga

```
data Kahend a
  = Tühi                          .
  | Tipp a (Kahend a) (Kahend a)
```
(128)

Tüübiparameeter definitsioonis (128) vastab puus hoitavate andmete tüübile. Esimene alternatiiv kirjeldab tühja kahendpuu, teine mittetühja. Teise alternatiivi konstruktori esimese argumendi kohal on tüübiparameeter, mis vastab andmele selles tipus. Ülejäänud kaks argumenti on rekursiivsed pöördumised, mis tähendab, et puu need väljad on ise sama tüüpi puud. Loeme näiteks esimese neist vasakuks ja teise paremaks alampuuks.

Näiteks on korrektsed järgmised `Kahend`-tüüpi avaldised: `Tühi`, `Tipp 0 Tühi Tühi`, `Tipp 5 (Tipp 2 Tühi (Tipp 3 Tühi Tühi)) (Tipp 7 Tühi Tühi)`.

Standardne valik puhuks, kus andmed paiknevad ainult kahendpuu lehtedes, on definitsiooniga

```
data Kahend' a
  = Leht a
  | Harud (Kahend' a) (Kahend' a)
```
(129)

antud tüübid. Tüübiparameetri roll on sama mis `Kahend`-tüübi korral. Esimene alternatiiv kirjeldab ühetipulised puud, neis on igaühes üks väärtus. Teine alternatiiv vastab vahetippudele nagu definitsioonis (128), kuid nüüd puudub komponenditüüpi väli. Seega vahetippudesse ei saa andmeid paigutada.

Definitsioon (129) kitsendab muuski mõttes kahendpuude hulka. Nimelt on välistatud tühi puu, mis omakorda tingib selle, et igal tipul on parajasti kas 0 või 2 alluvat.

Kahendpuu komponentide kokkuarvutamine pole sugugi keerulisem kui lineaarse struktuuri korral. Sissetoodud kahendpuutüüpide jaoks võiksime komponentide summat arvutavad funktsioonid kirjeldada koodidega

```
kahendSumma
  :: (Num a)
  => Kahend a -> a
kahendSumma (Tipp x ut vt)
  = x + kahendSumma ut + kahendSumma vt
kahendSumma _
  = 0
```
, (130)

```
kahend'Summa
  :: (Num a)
  => Kahend' a -> a
kahend'Summa (Harud ut vt)
  = kahend'Summa ut + kahend'Summa vt
kahend'Summa (Leht x)
  = x
```
. (131)

Paljuharulise puu tüübina kasutatakse funktsionaalses programmeerimises tavaliselt nn **Rose'i puud**, mis sisaldab juurtipu ning tema 0 või enam kindlas järjekorras alampuud, millest igaüks

on omakorda Rose'i puu. Alampuude järjend esitatakse listina. Siit tuleneb definitsioon

```
data Rose a
  = Juur a [Rose a].
```
(132)

See, et definitsioonis (132) puudub rekursiivse pöördumiseta alternatiiv, ei tähenda, et Rose'i puud alati lõpmatuseni hargnevad. Kui alampuude list on tühi, siis alampuid järelikult pole ja tegemist on lehega, alluvateta tipuga. Nii on korrektsed avaldised näiteks `Juur 0 []`, `Juur 1 [Juur 2 [], Juur 3 []]` jpt. Hargnemine ja mittehargnemine on kirjeldatud ühtsel viisil.

Rose'i puu komponentide summa arvutamine on ka peaaegu niisama lihtne kui kahendpuul, sobib kood

```
roseSumma
  :: (Num a)
  => Rose a -> a                .
roseSumma (Juur x rs)
  = x + sum (map roseSumma rs)
```

Rose'i puu alampuud moodustavad kokku Rose'i metsa. Tihti on Rose'i puudega tegelemisel mugav Rose'i metsa tüübile eraldi nimi anda. Selleks võiksime sisse tuua tüübisünonüümi `Mets` koodiga

```
type Mets a
  = [Rose a].
```
(133)

Seda nime kasutades võib lihtsustada definitsiooni (132), kirjutades

```
data Rose a
  = Juur a (Mets a).
```
(134)

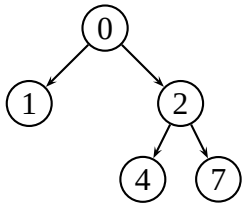
Nüüd on uue tüübi definitsioon (134) tüübisünonüümi definitsiooniga (133) vastastikku rekursiivne.

Vastastikku tohivad rekursiivsed olla ka uue tüübi definitsioonid omavahel. Haskell ei luba aga tüübisünonüüme isekeskis rekursiivselt defineerida. Tüübitaseme definitsioonide rekursiivsete pöördumiste tsüklites peab alati sees olema uue tüübi definitsioon.

Ülesandeid

335. Kirjutada oma moodulisse jorutüübi deklaratsioon (127). Defineerida muutuja `joruTake` väärtuseks funktsioon, mis töötab jorudel nagu `take` väärtuseks olev funktsioon listidel.

336. Kirjutada avaldis tüüpi `Kahend Int`, mis väljendab joonisel



kujutatud kahendpuud.

337. Kirjutada mõned avaldised, mis väljendavad erineva suurusega `Kahend'`-tüüpi kahendpuid.

338. Defineerida muutuja `kahendSuurus` väärtuseks funktsioon, mis võtab argumendiks kahendpuu deklaratsiooniga (128) defineeritud tüübist ja annab tulemuseks tema komponentide arvu. Defineerida sarnaselt muutujad `kahend'Suurus` tüübipere `Kahend'` jaoks ning `roseSuurus` Rose'i puude jaoks.

339. Defineerida puutüüp, mille igal puul on juurtipp, iga puu igal tipul on kas 0 või 2 alluvat ning igas tipus on andmekirje.

340. Kirjutada operaatorid ülesandes 339 defineeritud kahendpuu andmeväljade arvu ja summa leidmiseks.

341. Defineerida Rose'i puu ja Rose'i metsa tüübid nii, et metsatüüp oleks uus ja puutüüp sünonüüm. Üldine struktuur peab säilima, liigendkohtade konkreetne ehitus võib olla teistsugune kui ülaltoodud definitsioonide puhul.

342. Kirjutada operaatorid ülesandes 341 defineeritud puude andmeväljade arvu ja summa leidmiseks.

Erinevate andmestruktuuritüüpide paralleelsel kasutamisel on tüüpiliseks ülesandeks nende omavaheline konverteerimine. Kuna list on kõige kasutatavam struktuur, on eriti tihti vaja erinevaid mittelineaarseid struktuure lamendada (ingl *flatten*), nendes hoitavad komponendid listi ümber paigutada, kaotades algse struktuuri, aga samuti vastupidist teisendust, mittelineaarse andmestruktuuri sisselugemist listi pealt.

Lamendamine seisneb sisuliselt andmestruktuuri läbimise viisi määramises. On olemas kaks klassikalist rekursiivset puuläbimisviisi: eesjärjestus ja lõppjärjestus. Eesjärjestuse (ingl *pre-order*) puhul võetakse algul ette puu juur, seejärel järjest iga alampuu tipud eesjärjestuses. Lõppjärjestuse (ingl *post-order*) puhul läbitakse kõigepealt järjest iga alampuu tipud lõppjärjestuses ja lõpuks juur. Mõlemal juhul võetakse alampuud ette samas järjestuses. Kahendpuu juures tuntakse kolmanda klassikalise läbimisviisina keskjärjestust (ingl *in-order*), mille korral juur läbitakse vasaku ja parema alampuu läbimise vahel, kumbki alampuu läbitakse omakorda keskjärjestuses.

Realiseerime kõik need läbimisviisid Haskellis `Kahend`-tüüpide peal. Otse määratluse järgi kirjutades saaksime näiteks eesjärjestuse puhul naiivse definitsiooni

```
läbiEes
  :: Kahend a -> [a]
läbiEes (Tipp x ut vt)
  = x : läbiEes ut ++ läbiEes vt
läbiEes _
  = []
```
(135)

See aga on ilmselt ebaefektiivne, sest konkatenatsiooni vasakus argumendis esineb rekursiivne pöördumine. Selle asemel tuleb ehitada tulemuslist akumulaatoris, tegemist on akumulaatori kasutamise standardjuhuga. Niisugusel lähenemisel saame koodi

```
läbiEes tt
  = let
      läbiEes (Tipp x ut vt) as
        = x : läbiEes ut (läbiEes vt as)
      läbiEes _              as
        = as
    in
    läbiEes tt []
```

Samal põhimõttel saame lõppjärjestuse ja keskjärjestuse leidmiseks koodid

```
läbiLõpp
  :: Kahend a -> [a]
läbiLõpp tt
  = let
      läbiLõpp (Tipp x ut vt) as
        = läbiLõpp ut (läbiLõpp vt (x : as))
      läbiLõpp _              as
        = as
    in
    läbiLõpp tt []
```

```
läbiKesk
  :: Kahend a -> [a]
läbiKesk tt
  = let
      läbiKesk (Tipp x ut vt) as
        = läbiKesk ut (x : läbiKesk vt as)
      läbiKesk _              as
        = as
    in
    läbiKesk tt []
```

Näitena puustruktuuri konstrueerimisest listi pealt “kehastame” nüüd kahendpuudel selle skeemi, millega sai eespool defineeritud listi järjestamine põimimismeetodil ja mida sai seal nimetatud listi kahendpuukujuliseks kokkuarvutamiseks. Olgu öeldud, et lisastruktuuri kasutamise tõttu töötab uus realisatsioon pisut aeglasemalt kui endine (keerukus siiski endiselt halvimal juhul $O(n \log n)$ ja parimal $O(n)$); näite ainus eesmärk on selgitada paralleele kahendpuukujulise kokkuarvutamise ja tõeliste defineeritud kahendpuude vahel.

Kasutame siin deklaratsiooniga (129) sissetoodud kahendpuutüüpe, kus andmed asuvad ainult lehtedes. On vaja programmeerida operatsioon, mis listi pealt sellise kahendpuu üles ehitab, nii et listi elemendid jääksid tema lehtedesse. Teeme seda kahes osas, mis vastavad operaatoritele `põimiKahekaupa` ja `põimiPuu` eespool.

Esimeses osas on vaja kirjeldada funktsioon, mis lisab ehitatavasse struktuuri ühe taseme. Vahestruktuurina sobib kasutada puude listi: see sisaldab elemendina need puud, mis on parasjagu käsutada, ja operatsioon seisneb nende kahekaupa kokkusidumises suurematеks puudeks, mis vähendab puude arvu ümmarguselt kaks korda. Saame koodi

```
kahekaupa
  :: [Kahend' a] -> [Kahend' a]
kahekaupa (xt : yt : yts)
  = Harud xt yt : kahekaupa yts
kahekaupa xts
  = xts
```

Teise osana peame kirjeldama funktsiooni, mis ühe taseme lisamist itereerib, nii et lõpuks üksainuke puu kõik elemendid endasse haarab. Lisame lõppu ka operatsiooni, mis selle puu listist välja võtab. Saame koodi

```
puu
  :: [Kahend' a] -> Kahend' a
puu xts@ (_ : _ : _)
  = puu (kahekaupa xts)
puu [xt]
  = xt
```

Kuna erinevalt kahendpuukujulisest kokkuarvutusest listidel me nüüd ei põiminud, vaid ehitasime ainult puustruktuuri, siis tuleb põimida tagantjärele. Põimimist kahendpuul on aga väga lihtne programmeerida, see on ülesehituselt sama mis koodiga (131) defineeritud elementide kokkuliit-

mine, ainult liitmisoperatsiooni rollis on põim. Sobib kood

```
üldPõim
  :: (Ord a)
  => Kahend' [a] -> [a]
üldPõim (Harud ust vst)                ,
  = põimi (üldPõim ust) (üldPõim vst)
üldPõim (Leht xs)
  = xs
```

kus muutuja `põimi` väärtuseks on eespool defineeritud kahe listi põimimine.

Nüüd on jäänud vaid järjestamine kirjeldada koodiga

```
järjestaPõim xs
  | null xs
    = xs                                  .
  | otherwise
    = üldPõim (puu (map Leht (hakid xs)))
```

Kui list on tühi, pole vaja midagi teha, lihtsalt anname tühja listi välja; muidu aga hakime listi järjestatud juppideks (operaator `hakid`), teeme iga jupi ühetipuliseks puuks (`map Leht`), ühendame need kokku üheks suureks puuks (`puu`) ja lõpuks põimime (`üldPõim`).

Ülesandeid

343. Defineerida muutuja `joruToList` väärtuseks funktsioon, mis võtab argumendiks joru deklaratsiooniga (127) defineeritud tüübist ja annab tulemuseks selle järjendi listina.

344. Defineerida muutuja `lõpmatuJoru` väärtuseks mõni lõpmatu struktuur deklaratsiooniga (127) defineeritud tüübist.

345. Defineerida muutuja `täielik` väärtuseks funktsioon, mis võtab argumendiks täisarvu n ja kui see on mittenegatiivne, siis annab tulemuseks täieliku kahendpuu kõrgusega n, kus tippudes on nullid.

346. Defineerida muutuja `pügaKahend` väärtuseks funktsioon, mis võtab järjest argumendiks täisarvu n ja kahendpuu t ja annab tulemuseks kahendpuu, mille saab puust t nende osade mahalõikamisel, mis jäävad juurest kaugemale kui n taset.

347. Defineerida muutuja, mille väärtuseks on selline lõpmatu kahendpuu, et rakendades sellele muutujale ülesandes 346 defineeritud operaatorit `pügaKahend`, saame lahendada ülesande 345.

348. Lahendada ülesandega 346 analoogiline ülesanne Rose'i puude jaoks.

349. Kirjutada operaatorid, millega saaks teisendada Rose'i puude ja metsade kahe esituse vahel: üks, mis antud deklaratsioonidega (133) ja (134), ja teine, mis kirjutatud ülesandes 341.

350. Defineerida muutuja `roseEes` väärtuseks funktsioon, mis võtab argumendiks Rose'i puu deklaratsiooniga (134) defineeritud tüübist ja annab tulemuseks listi, millesse on salvestatud selles puus olevad elemendid eesjärjestuses. Defineerida sarnane muutuja `roseLõpp` lõppjärjestuse jaoks.

351. Lahendada ülesandega 350 analoogne ülesanne Rose'i puude jaoks, mis on defineeritud ülesandes 341.

352. Defineerida muutuja `teed` väärtuseks funktsioon, mis võtab argumendiks Rose'i puu ja annab tulemuseks sama tüüpi komponentidega listide listi, kus listid vastavad teedele juurest lehte, listi elementideks teel kohatud andmed.

Indeks