

Haskelli interaktiivsete interpretaatorite kasutamine

Haskell on funktsionaalne programmeerimiskeel, mille peal toimub käesoleva kursuse praktiline õppetöö.

Haskell-kood jaotatakse kõige välimisel tasemel mooduliteks (Java klassi analoog selles mõttes, mis puudutab koodi struktureerimist väiksemateks osadeks). Haskell-moodulite standardne laiend on `.hs`. Failinimi ilma laiendita langeb kokku selles defineeritava mooduli nimega; failinimed transformeeruvad moodulinimedeks koos oma suhtelise teega teegi põhikataloogi suhtes, kusjuures eraldajaks on kaldkriipsu asemel punkt. Näiteks standardteegi moodul `Data.Char` asub standardteegi põhikataloogi suhtes failis `Data/Char.hs`. Tsentraalne moodul on `Prelude`, mis asub standardteegi põhikataloogis failis `Prelude.hs` ja millega tulevad kaasa kõige põhilisemad eeldefineeritud ja sisse ehitatud asjad.

Hugs

Hugs on Haskell'i interpretaator, tema koduleht asub aadressil

<http://www.haskell.org/hugs/>.

Terminali käsurealt käivitub Hugs käsuga `hugs`. Hugsis saab interaktiivselt lasta väärtustada Haskell-avaldisi, lihtsalt sisestades neid käsurealt. Lisaks saab anda ka Hugs'i oma käsked, millest peamised on järgmised:

- `:q` väljumine,
- `:e` tekstiredaktori avamine,
- `:l` mooduli sisselugemine (argumendiks failinimi),
- `:r` viimase sisselugemiskäsu kordamine,
- `:t` avaldise tüübi kuvamine (argumendiks avaldis),
- `:i` info identifikaatorite kohta (argumendiks identifikaatorite loend).
- `:?` Hugs'i käskude nimekiri,
- `:s` eelistuste ja parameetrite nimekiri ja muutmine.

Tekstiredaktorit ei pea muidugi Hugsist käivitama. Oma programmid võib teha valmis ükskõik milliste vahenditega ja siis Hugsiga sisse lugeda. Hugs'i `:e` on hea selle poolest, et kui editorist väljuda, sooritab Hugs automaatselt ka kompileerimise. Aga täiesti normaalne on ka variant, kus

ühes aknas jookseb Hugs ja teises tekstiredaktor; viimases teeb kasutaja muudatusi ja loeb neid Hugi aknas sisse.

Käsk `:r` on kasulik, kui käib programmifaili muutmine, siis iga kord loetakse samast failist värske versioon sisse.

Lisaks käsule `:s` saab Hugi parameetrite vaikeväärtusi sättida ka keskkonnamuutujaga `HUGSFLAGS`. Näiteks minul on kodukataloogi failis `.login` rida

```
setenv HUGSFLAGS "-Eemacs -P.:${MYHSLIB}:${HUGSLIB} -h24M +kls".
```

Sealhulgas `-Eemacs` seab tekstiredaktoriks Emacsi ja `-P...` failiotsimistee, kus `MYHSLIB` ja `HUGSLIB` on minu defineeritud keskkonnamuutujad, mille väärtuseks on vastavalt mu enda Haskell-teekide ja Hugi teekide kataloogid.

Hugi standardteek ülikooli arvutivõrgu failisüsteemis asub kataloogis

```
/opt/local/lib/hugs/libraries.
```

Põhimoodul `Prelude` loetakse Hugi käivitamisel automaatselt sisse.

Ülesandeid

1. Käivitada Hugs.
2. Teises aknas visata silm peale Hugi standardteekidele.
3. Lugeda Hugsis vabal valikul sisse mõni moodul standardteegist; see võiks olla nt `Data.Char`, mida edaspidi niikuinii vaja tuleb (kuna see moodul on Haskell-98 standardis, saab ta kätte ka ilma paketinimeta `Data`).
4. Lasta Hugsil väärtustada lihtsaid Haskell-avaldisi, nt mõni aritmeetiline avaldis nagu `2 + (-3)`, `pi` vms.
5. Lasta Hugsil kuvada lihtsate avaldiste tüüpe.
6. Lasta Hugsil anda infot mõne teile tuntud identifikaatori kohta.
7. Lahkuda Hugsist.

GHCi

GHC on Haskell'i kompilaator, mille koduleht asub aadressil

<http://www.haskell.org/ghc/>.

GHC on arendatud Glasgow ülikoolis (GHC on lühend nimetusest '*Glasgow Haskell Compiler*'). Interaktiivne keskkond kannab nime GHCi ('GHC *interactive*').

GHCi käivitub terminali käsurealt käsuga `ghci`. Standardteegid on saadaval vaid kompileeritud kujul — ülikooli arvutivõrgus asuvad nad kataloogis `/opt/local/lib/ghc-<version>/imports/` —, algtekste näha ei saa. Hugs'i ja GHC teekide algtekstid on siiski valdavalt samad, kuna pärinevad samast repositooriumist.

Interaktiivse keskkonna käsud `:q`, `:t`, `:i`, `:?`, `:s` on sarnased Hugsiga. Standardteegi moodulite sisselugemisel tuleb kasutada käsku `:m`, oma moodulite puhul aga `:l` ja `:r` nagu Hugsis.

Ülesandeid

8. Proovida kätt ka GHCi peal.

Haskell'i süntaksi põhielemendid

Haskell-programmi osad klassifitseeruvad, peale võtmesõnade ja -sümbolite, peamiselt avaldis-tekst, näidisteks ja deklaratsioonideks.

Avaldisega (ingl *expression*) esitatakse üks väärtus olemasolevate muutujate väärtuste kaudu. Näiteks kui x väärtus on 2, y väärtus on 3 ja $+$ väärtus on liitmine, siis avaldise $x + y$ väärtus on $2 + 3 = 5$. Lihtsaimal juhul on avaldis lihtsalt muutuja, nt muutuja `pi` on ühtlasi avaldis.

Näidis (ingl *pattern*) kannab avaldisega duaalset eesmärki esitada ühe väärtuse kaudu muutujate väärtused. Näiteks kui näidise (a, b) väärtus on paar $(2, 3)$, siis a väärtus on 2 ja b väärtus on 3. Muutujate defineerimiseks kasutataksegi Haskellis näidiseid. Ka näidis koosneb lihtsaimal juhul lihtsalt ühest muutujast.

Et vältida võimalikke vastuolusid muutujate väärtuste leidmisel, ei tohi ükski näidis sisaldada ühtki muutujat mitu korda. Nt näidis (a, a) on illegaalne, sest nt väärtuse $(2, 3)$ puhul pole muutuja a väärtus üheselt määratav.

Deklaratsiooni (ingl *declaration*) ülesanne on midagi uut defineerida. Enamasti antakse deklaratsiooniga väärtus ühele või mitmele uuele andmemuutujale ning selline deklaratsioon saadakse avaldise ja näidiseid sobivalt kombineerides.

Programmitekst on deklaratsioonide kogum.

Lihtsamad avaldised

Haskellis on väga palju erinevaid avaldiste liike. Lihtsamad, mida vaatame siin jaotises, ei sisalda näidiseid ega deklaratsioone; hiljem jõuame ka keerulisemateni.

Nagu juttu oli, on avaldise üheks erijuhuks muutuja. Haskellis leksika seab muutujanimedele piiranguid. Muutujanimed peavad kas koosnema tähtedest, numbritest ja alakriipsust, alates seejuures väiketähega, või koosnema sümbolitest

`+, -, *, /, ^, =, <, >, &, |, $, :, !, ?, ., %, \, @, ~, #,` (1)

mispuhul ta ei tohi alata kooloniga.

Muutujatele vastanduvad konstruktorid (ingl *constructor*), mis on, nagu muutujadki, atomaarsed avaldised (st avaldised, mis ei koosne väiksematest süntaktilistest objektidest), kuid kujutavad endast lõppväärtust, mida enam edasi ei teisendata. Sellised on näiteks numbritega kirjutatud arvulised konstandid `0`, `1`, `1.5` jne, sümbolkonstandid `'A'`, `'z'` jne, tõeväärtused `True` ja `False` ning tühja listi märkiv `[]`. Kõik need on eeldefineeritud moodulis `Prelude`.

Haskellis leksikareeglite kohaselt konstruktorinimed, mis pole erikujulised, algavad suurtähe või kooloniga. Tavakujulised konstruktorid on eelmises lõigus vaadelduist tõeväärtused, ülejäänud on sisseehitatud erikujulised.

Vaatame edasi avaldiste moodustamist infiksoperaatoritega. Eelmises jaotises tõenäoliselt proovisite juba interaktiivses keskkonnas väärtustada aritmeetilisi avaldiseid nagu nt `2 + (-3)`, `2 * pi / 3`, kus avaldised on konstrueeritud atomaarsetest avaldistest infiksoperaatorite `+`, `*`, `/` abil. Nimetus 'infiksoperaator' tuleb sellest, et nende loomulik kasutus on infiksne, st oma argumentide vahel. Haskellis leksikareeglite kohaselt on identifikaatori loomulik kasutus infiksne, kui ta koosneb sümbolitest nimekirjas (1).

Astendamiseks on olemas kaks infiksoperaatorit `^` ja `**`; neist esimene sobib juhul, kui astendaja on naturaalarv, teine on ujukomaarvude astendamiseks. Standardteegi moodulis `Data.Ratio` defineeritud infiksoperaator `%` võtab kaks täisarvu ja annab välja nende jagatise ratsionaalarvuna. Ratsionaalarvude lõppkuju defineeritakse seal kui esitus taandumatu murruna.

Ka võrdlusoperatsioonid on infiksoperaatorid. Nende abil saame 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.

Loogiliste avaldiste tegemiseks on kasutatavad infiksoperaatorid `&&` ja `||`, mille väärtusteks on vastavalt konjunktsioon ja disjunktsioon. Väärtustamist alustatakse vasakust argumentist. Kui `&&` vasak argument on väär, siis paremat argumenti ei väärtustata. Kui `||` vasak argument on tõene, siis paremat ei väärtustata.

Infiksoperaatoritega seonduvad atribuudid prioriteet ja assotsiatiivsus. Kõrgema prioriteediga operatsioonid tuleb sulgudeta avaldises teha enne madalama prioriteediga operatsioone. Arit-

meetiliste operatsioonide prioriteetide järjekord on sama mis matemaatikas, st astendamised on kõrgema prioriteediga kui korrutamine ja jagamine, mis omakorda on kõrgema prioriteediga kui liitmine ja lahutamine. Võrdlusoperatsioonid on aritmeetilistest operatsioonidest madalama prioriteediga. Loogilistest operatsioonidest on $\&\&$ kõrgema prioriteediga kui $|$ ning nad mõlemad on madalama prioriteediga isegi võrdlusoperatsioonidest.

Infiksoperaator saab olla vasak-, parem- või mitteassotsiatiivne. Vasakassotsiatiivse operaatori esinemisi tuleb rakendada vasakult paremale, paremassotsiatiivse operaatori esinemisi paremalt vasakule. Näiteks astendamine on paremassotsiatiivne, seega avaldis $9 \wedge 9 \wedge 9$ on ekvivalentne avaldisega $9 \wedge (9 \wedge 9)$, mitte avaldisega $(9 \wedge 9) \wedge 9$. Mitteassotsiatiivsete operatsioonide mitmekordse esinemise puhul on järjekorra näitamine sulgudega kohustuslik.

Prioriteete märgivad Haskellis arvud 0-st 9-ni, kõrgemaid prioriteete tähistavad suuremad arvud. Käsk `:i` interaktiivses interpretaatoris annab infiksoperaatorite puhul teada lisaks tüübile ka prioriteedi ja assotsiatiivsuse.

Ülesandeid

9. Kirjutada interaktiivse interpretaatori käsurealt avaldis, mis kontrollib, kas $\pi \leq \frac{22}{7} < \sqrt{10}$. Kasutada võimalikult vähe sulgusid.

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

Äsjavaadeldutega sarnasel kujul on näiteks avaldis $(2, -3)$, mille väärtuseks on paar $(2, -3)$. Paar võib olla ka keerulisemate komponentidega, nt $(1 + 1, 2 - 5)$, mille väärtus on samuti $(2, -3)$. Paari komponendid ei pea olema üht tüüpi, nt $('A' , False)$ on korrektne avaldis.

Paneme tähele, et mis iganes avaldised paari komponentideks ka ei ole, tema väärtus on ikka paar. Sulud ja koma arvutuse käigus ära kaduda ei saa. Seetõttu on paarimoodustamise näol tegemist konstruktoriga. Erinevus ülalvaadeldud konstruktoritega on, et need ei omanud argumente, paarimoodustamine aga kujutab endast binaarset operaatorit.

Vaatame lisaks operatsioonide infikssele rakendamisele ka prefiksset rakendamist, mis tähendab, et rakendatav funktsioon asumas oma argumenti ees, mis on isegi levinum kui infiksne kasutus. Haskellis leksikareeglite kohaselt on identifikaatori loomulik kasutus prefiksne, kui ta koosneb tähtedest, numbritest ja alakriipsust.

Haskellis eraldatakse funktsioon oma argumentid üldiselt tühikuga, nt kujul `log 5`. Tühiku asemel on lubatud ka muu mittetühi tühisümbolite jada. Kui kas funktsioon või argument on sulgudes, võib vahet ka mitte olla, nt variandid `log(5)`, `(log)5`, `(log)(5)` on kõik korrektsed ja tähendavad sama mis `log 5`.

Prelüüdis on defineeritud väga palju väga erinevaid funktsioone, mille tavakasutus on prefiksne. Näiteks matemaatilised funktsioonid `sin` (siinus), `cos` (koosinus), `tan` (tangens), `asin` (arkus-

siinus) jne, `log` (naturaallogaritm), `exp` (eksponentfunktsioon alusel e), `abs` (absoluutväärtus) jms, loogiline funktsioon `not` (eitus) jpm. Moodulis `Data.Char` on veel `ord`, mis leiab sümboli koodi, tema pöördfunktsioon `chr`, mis leiab koodi järgi sümboli, `isUpper`, mis kontrollib, kas argumentsümbol on suurtäht, `toUpper`, mis teisendab sümboli, kui see oli väiketäht, vastavaks suurtäheks, veel sümboli klassikuuluvust kontrollivaid ja teisendavaid ning muid funktsioone.

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 funktsioon ja tema argument, siis selle operaatori prioriteet on 10 ja ta on vasakassotsiatiivne.

Ülesandeid

11. Kontrollida ujukomaarvuliste arvutuste täpsust, avaldades arvu π mõne arkusfunktsiooni kaudu ja võrreldes muutuja `pi` väärtusega.
12. Kompleksmuutuja funktsioonide teoorias defineeritakse $i^i = e^{-\frac{\pi}{2}}$. Arvutada see arv interaktiivse interpretaatoriga.
13. Kontrollida ühe avaldise väärtustamisega, kas sümbol koodiga `7` on suurtäht.

Funktsioon `show` teisendab oma argumendi stringiks. See funktsioon on defineeritud arvudel, sümbolitel, tõeväärtustel ja paljude teiste tüüpide esindajatel, kuid mitte kõigil Haskellis ette tulevatel objektidel. Näiteks funktsioonide stringiks teisendamine on standardteegis eeldefineerimata. Samuti võib juhtuda, et `show` on küll defineeritud, kuid mingil põhjusel nii, et tulemus ei kajasta päris täpselt väärtuse originaalset struktuuri. Näiteks ratsionaalarvude lõppkuju taandumatu murruna kasutab konstruktorit `%`, kuid ratsionaalarvulised väärtused teisendatakse stringikujule ilma koolonita, justnagu `%` oleks konstruktor.

Funktsiooni `show` lisab interaktiivne interpretaator kõigile käsurealt antud avaldistele (tõsi, mitte sisend-väljund-tüüpi avaldistele, kuid neid me pole veel õppinud), et arvutuse tulemust väljastada.

Ülesandeid

14. Anda interaktiivse interpretaatori käsurealt avaldis, milles funktsioon `show` on rakendatud mõnele arvule, sümbolile või tõeväärtusele ilmutatult. Mis vahe on saadavas vastuses võrreldes sellega, kui `show` on puudu? Selgitada.

Paaridega seonduvad standardfunktsioonid `fst` ja `snd`, mis paarile rakendades annavad vastavalt esimese ja teise elemendi. Standardfunktsioonidest annab paari välja näiteks `properFraction`, mis võtab argumendiks ujukomaarvu ja annab väärtuseks paari tema täis- ja murdosaga.

Ülesandeid

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

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 paremseltsioonideks. Vasakseksioon (ingl *left section*) näeb välja kujul $(a \oplus)$, kus a on avaldis ja \oplus on infiksoperaator. Sellise avaldise väärtus on funktsioon, mis iga argumendi y korral annab väärtuseks $\bar{a} \oplus y$, kus \bar{e} tähistab e väärtust. Analoogselt näeb paremseltsioon välja kujul $(\oplus b)$, kus \oplus on infiksoperaator ja b avaldis. Sellise avaldise väärtus on funktsioon, mis iga argumendi x korral annab väärtuseks $x \oplus \bar{b}$.

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 väärtustub näiteks $(* 5) 7$ arvuks 35, $(3 ^) 2$ arvuks 9 ja $(^ 3) 2$ arvuks 8.

Vasakseksioone saab moodustada kõigist infiksoperaatoritest. Paremseltsioone saab moodustada kõigist infiksoperaatoritest peale miinuse, sest kirjutis kujul $(- a)$ läheks segi negatiivse arvuliteaaliga $-a$.

Ülesandeid

16. Proovida interaktiivse interpretaatori käsurealt rakendada mõne mittekommutatiivse infiksoperaatori vasak- ja paremseltsiooni.
17. Selgitada kirjutise $(: [])$ sisu.

Listiavaldised

Listidega tegelemisel on tarvis infiksoperaatorit $:$, mis oma vasaku argumendi lisab parema argumendina saadud listi ette. Listi esimest elementi nimetatakse listi peaks (ingl *head*), ülejäänud elementide listi listi sabaks (ingl *tail*). Niisiis $:$ eraldab mittetühja listi puhul tema pea ja saba. Kuna list Haskellis on üht tüüpi objektide ahelalmestruktuur, siis $:$ vasak argument peab olema sama tüüpi mis parema argumendina saadud listi elemendid, muidu tekib tüübiviga. Näiteks $2 : []$ väärtuseks on list, mille ainus element on 2, $1 : 3 : 5 : 7 : 9 : []$ väärtuseks aga 5-elementiline list, mille elemendid on järjest 1, 3, 5, 7, 9. Viimasest näitest nähtub, et operaator $:$ on paremassotsiatiiivne, st sulgudeta kirjutis $a : b : c$ tähendab sama mis $a : (b : c)$.

Seejuures ka `:` on konstruktor, teda arvutusprotsess ei puutu.

Ülesandeid

18. Selgitada välja operaatori `:` prioriteet. Kuhu ta paigutub senituntud infiksoperaatorite prioriteetide järjestuses?
19. Lasta interpretaatori käsurealt väärtustada 3-elementiline list, mille elemendid on baitide arvud vastavalt kilobaidis, megabaidis ja gigabaidis.
20. Lasta interpretaatori käsurealt väärtustada 2-elementiline list, mille esimene element ütleb, kas $\tan 27^\circ > 0.5$, ja teine, kas $30 < \pi^3 \leq 31$.
21. Kirjutada interpretaatori käsurealt mittetühi list, mille elemendid on omakorda listid.

Kuna listid on praktilises funktsionaalses programmeerimises põhilised andmestruktuurid, on Haskellis nende jaoks sisse ehitatud spetsiaalsed süntaktilised konstruktsioonid, mis tihti lubavad liste kirjutada lühemalt kui võimaldaksid üksi konstruktorid `:` ja `[]`.

Põhilisim neist on listi esitamine komadega eraldatud elementide loeteluna kantsulgudes. Näiteks ülalvaadeldud listid `2 : []` ja `1 : 3 : 5 : []` on selles notatsioonis vastavalt `[2]` ja `[1, 3, 5, 7, 9]`. Funktsioon `show` tekitab listidest tüüpiliselt just sellise esituse.

Veel on kasutatav erisüntaks aritmeetilise progressiooni esitamiseks. Üldine progressioon esitatakse kahe esimese elemendi ja piiri kaudu kujul `[a, b .. c]`: selle kirjutise väärtus on list, mille elemendid võetakse järjest aritmeetilisest jadast esimese elemendiga \bar{a} vahega $\bar{a} - \bar{b}$ parajasti senikaua, kui elemendid jäävad arvust \bar{c} samale poole kui arv \bar{a} . Näiteks võinuksime listi `[1, 3, 5, 7, 9]` samaväärselt kirjutada kujul `[1, 3 .. 9]`, samuti kujul `[1, 3 .. 10]`. Kui progressioon on sammuga 1, võib teise elemendi ära jätta. Näiteks avaldise `[1 .. 10]` väärtuseks on list järjekorras arvudest 1 kuni 10.

Ülesandeid

22. Koostada aritmeetilise jada süntaksiga avaldise ja anda neid interaktiivse interpretaatori käsurealt. Proovida nii positiivse kui negatiivse vahega progressioone ning kummalgi juhul nii esimesest elemendist suurema kui väiksema väärtusega piiriga.
23. Mis saab, kui vahe on 0? Lõpmatu arvutuse katkestab `<Ctrl>+c`.

Haskellis võib legaalse avaldise väärtus olla ka lõpmatu andmestruktuur. Sellise avaldise täielik väärtustamine jääb tsüklisse, kuid laisk väärtustamine teeb võimalikuks sellise avaldise valutu kasutamise, kui kontekst on selline, mis ei nõua tema täielikku väärtustamist. Iga listi struktuur

ehitatakse mällu ainult niikaugemale, kui programmi täitmise seisukohast tarvilik. Seega kui lõpmatu listil läheb reaalselt vaja vaid lõplikku algusjuppi, võib lõpmatuid liste kirjeldav programm normaalselt oma töö lõpetada. Lõpmatud listid on head programmeerimise mugavuse mõttes, sest ülesande seisukohalt kunstliku piiri sissetoomine teeks koodi tunduvalt kohmakamaks.

Lõpmatut listi on kõige lihtsam tekitada aritmeetilise jada süntaksiga, jättes piiri ära. Näiteks `[1, 3 ..]` väärtuseks on list järjekorras kõigi paaritute naturaalarvudega, `[1 ..]` väärtuseks aga list järjekorras kõigi positiivsete naturaalarvudega.

Haskellis programmeerides puutume tihti kokku mitte ainult eraldiseisvate lõpmatute listidega, vaid andmestruktuuridega, mille komponentideks on lõpmatud listid. Näiteks `([1, 3 ..] , [2, 4 ..])` 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.

Stringid on Haskellis sümbolite listid. Vastavus on selline, et tühi string kujutab endast tühja listi; mittetühja stringi pea on tema esimene sümbol ning saba tema alamstring alates teisest sümbolist kuni lõpuni. Seda tasub arvestada, kui stringe kasutavaid funktsioone defineeritakse. Stringide tavapärane kirjalpilt katkematu sümbolite joruga jutumärkides on Haskellis kasutatav, kuid see on sisseehitatud erisüntaks. Ka funktsioon `show` tekitab kõigist sümbolite listidest sellise kuju.

Sümbolite listidel on kasutatavad ka teised senivaadeldud erisüntaksid: kantsulgudes elementide komadega eraldatud loend ning aritmeetiline progressioon. Viimane baseerub kooditabelil: sümboleid loetakse nagu neile vastavaid koode.

Ülesandeid

24. Kirjutada interaktiivse interpretaatori käsurealt avaldise, mille väärtuseks on string "Tere!". Kasutada erinevaid süntaktilisi konstruktsioone.
25. Kirjutada interaktiivse interpretaatori käsurealt võimalikult lühike avaldis, mille väärtuseks oleks ladina tähestik stringina.

Prelüüdis defineeritakse veel üks listidega seonduv infiksoperaator, nimelt `++` — konkatenatsioon. Kui `a` ja `b` väärtusteks on sama tüüpi elementidega listid, siis `a ++ b` väärtus saadakse `a` kopeerimisel `b` ette. Näiteks `[1, 2] ++ [1, 3, 6]` väärtuseks on list elementidega järjekorras 1, 2, 1, 3, 6. Seejuures arvutuse käigus ehitatakse parempoolse listi ette vasakpoolse koopiat, parempoolset listi ei uurita üldse. See tähendab, et konkatenatsiooni teostamine sõltub ainult vasakpoolse listi pikkusest. Sellel põhjusel on konkatenatsioon paremassotsiatiivne, sest nii on arvutus efektiivsem. Arvutades avaldist `(a ++ b) ++ c`, kopeeritakse kõigepealt list `a` listi `b` ette ja seejärel mõlemad koos listi `c` ette. Esimest listi kopeeritakse kaks korda. Sulgude

parempoolse paigutuse korral kopeeritakse iga listi ülimalt üks kord. Efektiivsuse vahe on seda suurem, mida rohkem liste järjest tuleb konkateneerida.

Listide puhul peaks tundma funktsioone `head` ja `tail`, mis annavad vastavalt argumentlisti pea ja saba, kui need eksisteerivad. Funktsioon `null` kontrollib, kas argumentlist on tühi. Need funktsioonid töötavad konstantse keerukusega, töö ei sõltu listi pikkusest.

Prelüüdis on listifunktsioonidele pühendatud väga suur osa. Kõige lihtsamad neist lisaks seniõpituile on `length`, `sum` ja `product`, mis arvutavad vastavalt oma argumentlisti elementide arvu, elementide summat ja elementide korrutist. Esimene neist on korrektselt rakendatav igat tüüpi elementidega listidele, teised kaks aga ainult arvulist tüüpi elementidega listidele. Veel on kasulik teada funktsiooni `reverse`, mis võtab argumendiks listi ja annab väärtuseks ümberpööratud listi. See funktsioon rakendub korrektselt mistahes tüüpi elementidega listidele. Kõik selles lõigus vaadeldud funktsioonid töötavad argumentlisti pikkuse suhtes lineaarses ajas.

Infiksoperaator `!!` võtab vasakpoolseks argumendiks listi, parempoolseks argumendiks täisarvu ning leiab antud listis antud järjekorranumbriga elemendi, kui selline on olemas. Seejuures elemente nummerdatakse alates 0-st, nii et `nt a !! 0` ja `head a` on iga listitüüpi `a` korral sama väärtusega. Listi viimase elemendi annab funktsioon `last`.

Kuna listi elemendid üldiselt ei paikne mälus kompaktselt, vaid lististruktuur ehitatakse mälus üles järjestikuste viitade ahelana, nii et listitüüpi muutuja viitab selle algusele, ei ole listi kaugemaid elemente võimalik konstantse keerukusega leida. Operaator `!!` töötab seepärast lineaarses ajas oma teise argumendi suhtes, `last` aga lineaarses ajas listi pikkuse suhtes.

Ülesandeid

26. Kirjutada interaktiivse interpretaatori käsurealt võimalikult lühike avaldis, mille väärtuseks oleks ühes stringis ladina tähestik suurtähtedest, millele järgneks ladina tähestik väiketähtedest.
27. Arvutada ladina tähestiku tähtede arv.
28. Arvutada kõigi 3-kohaliste 1-ga lõppevate arvude summa ja korrutis.
29. Leida suuruselt kolmeteistkümnes 2005-ga jaguv arv esimese 100000000 naturaalarvu seas. Kirjutada selleks niisugune avaldis, mis ei sisalda ühtki aritmeetikatehet.

Vigased avaldised

Avaldis võib olla vigane mitmel põhjusel. Mõni selles esinev identifikaator võib olla defineerimata. Selle avastab iga süsteem kohe ja ei hakkagi avaldist väärtustama. Seega on tegemist staatilise veaga. Põhimõtteliselt sama viga tekib, kui identifikaator on defineeritud moodulis, mis on sisse lugemata.

Olukorras, kus avaldise osad ei klapi omavahel tüübiliselt, näiteks funktsiooni argument pole seda tüüpi, mida funktsioon ootab, tekib tüübiviga. Haskellis on ka tüübivead staatilised, need avastatakse staatilisel analüüsil enne väärtustamisprotsessi alustamist. Kuna Haskellis tüübisüsteem on väga keeruline, on ka tüübivigu üsna mitut erinevat liiki.

Erinevalt C-st ja Javast ei tee Haskellis süntaks vahet identifikaatoritel, mille väärtuseks on funktsioon, ja identifikaatoritel, mille väärtuseks on näiteks arv. Kui funktsiooni kasutada kohas, kus peab olema arv, ei teki süntaksi-, vaid tüübiviga. Identifikaatorid, mille väärtuseks on funktsioon, nagu näiteks `log`, on süntaktiliselt tavalised muutujad.

Ülesandeid

30. Anda interaktiivse interpretaatori käsurealt avaldis, mis sisaldab (a) väiketähega algavat defineerimata identifikaatorit, (b) suurtähega algavat defineerimata identifikaatorit. Lugeda mõlemal korral veateadet ja saada aru.
31. Lugeda veateadet, mille interaktiivne interpretaator annab avaldisele `reverse 'A'`. See on üks võimalik liik tüübiviga — otsene tüübiviga. Püüda teate mõttest aru saada.
32. Lugeda veateadet, mille interaktiivne interpretaator annab avaldisele, kus mõni funktsioon on kohas, kus peaks olema arv, nt `asin log`. Selline on teine levinud tüübivealiik — tüübi vajalike omaduste mittetuletatavus. Võrrelda tüübiveateadet sellega, mis järgneb avaldisele `'A' + 1` — viga on sama liiki. Seega funktsioon funktsiooni argumendiks ei olnud siin määrav. Püüda ka siin asjast aru saada.

Mõnikord võib juhtuda ka, et interaktiivne interpretaator annab ilma väärtustamisele asumata veateate avaldisele, mis on täiesti korrektne. Ka see on olemuselt tüübiviga — see juhtub siis, kui avaldis on niisugust tüüpi, mille jaoks puudub stringikeisendamiskeskmine `show`. Kuna interaktiivne interpretaator lisab avaldistele omatahtsi selle funktsiooni, tekibki tüübiviga.

Ülesandeid

33. Anda interaktiivse interpretaatori käsurealt mõni paljas funktsioon ja jätta meelde tekkinud veateade: see järgneb alati juhul, kui väärtuse jaoks on funktsioon `show` defineerimata.

Kui funktsiooni `head` või `tail` rakendada tühjale listile, on avaldis küll igati korrektne, kuid väärtustamisel tekib täitmisaegne viga. Täitmisaegseid vigu saab programmeerija ka ise tekitada funktsiooniga `error`, mis saab argumendiks stringi ja lõpetab töö, andes selle stringi veateatena väljundisse.

Täpsemalt on siin tegemist nn sünkroonse (ingl *synchronous*) veaga, mis tähendab seda, et vea põhjus tulenes arvutusprotsessi sisemistest omadustest. Sellele vastanduvad asünkroonsed

(ingl *asynchronous*) vead, mille põhjused peituvad keskkonnas, nt mälu ületäitumine, protsessile saadetud signaal jms.

Teoorias ütleme, et `head` väärtus tühjal listil on `bottom`, samuti `tail` väärtus tühjal listil ning `error` väärtus mistahes stringil. Tüübiprobleeme ei ole, sest igal tüübil loetakse olevat oma `bottom` väärtuste hulgas sees.

`Bottom` on väärtuseks kõigile korrektsetele avaldistele, mille väärtustamise käigus ei teki avaldise välistasemele lõpliku ajaga ühtki konstruktorit. `Bottom` põhjus võib olla täitmise lõppemine sünkroonse veaga enne ühegi konstruktori väljatulekut nagu see on eelmise lõigu näidete puhul, aga samuti lõpmatu arvutus, mille jooksul ei teki iialgi ühtki konstruktorit välistasemele.

Konstruktori väljatulemine annab teatavat informatsiooni vastuse kohta, kuna ta on oma olemuselt püsiv, mistõttu on mõtet seda `bottom`ist eristada. Lõpmatu list ei ole `bottom`, sest tema väljaarvutamisel tekib kogu aeg uusi `:`-konstruktooreid. Enamgi, lõpmatu listi ükski alamlist ei ole `bottom`. Listi, mille mõni alamlist on `bottom`, nimetatakse osaliseks (ingl *partial*). Lõplikuks nimetatakse kõiki ülejäänud liste. Pöörame tähelepanu, et listi elementide kohta polnud siin midagi öeldud.

Ülesandeid

34. Anda interaktiivse interpretaatori käsurealt korrektne avaldis, mille väärtustamisel lõpetatakse töö teie ette antud veateatega. Võrrelda veateatega, mis järgneb `head []` väärtustamisele.
35. Kontrollida interaktiivse interpretaatoriga, et `head` ja `tail` töötavad ka lõpmatutel listidel.
36. Anda interaktiivse interpretaatori käsurealt avaldis, mille väärtus on list, mis pole ise `bottom`, kuid mille mingi alamlist on `bottom`. Mis juhtub, kui kästa arvutada sellise listi pea?
37. Anda interaktiivse interpretaatori käsurealt avaldis, mille väärtus on lõplik list, mille mõni element on `bottom`. Arvutada selle listi pikkus.
38. Milline on funktsioonide `length`, `sum`, `product`, `reverse`, last väärtus (a) tühjal listil, (b) lõpmatul listil, (c) osalisel listil?
39. Katsetades interaktiivse interpretaatoriga, uurida, mis on `++` väärtus listidel l_1 ja l_2 , kui (a) l_1 on lõpmatu, (b) l_1 on osaline, (c) l_1 on lõplik, kuid l_2 lõpmatu, (d) l_1 on lõplik, kuid l_2 osaline.
40. Kas materjalis toodud kirjeldus `!!` kohta peab paika, kui list on (a) lõpmatu, (b) osaline?
41. Nimetame funktsiooni agaralt väärtustavaks, kui mistahes argumendil välja kutsutult väärtustatakse kõigepealt argument kuni välimise konstruktori ilmumiseni. Demonstreerida interaktiivses interpretaatoris, et funktsioon `error` ei ole agaralt väärtustav.

Lihtsamad andmedeklaratsioonid, näidised

Selles jaotises algab juba tõeline programmeerimine — oma deklaratsioonide kirjutamine. Kõik selle ja järgmise jaotise deklaratsioonid võiks koondada moodulisse `Syntax.hs`.

Vaatleme selles peatükis ainult kõige põhilisemaid deklaratsioone, mis defineerivad andmemuutujaid. Peale nende on Haskellis võimalik defineerida ka tüübimuutujaid, uusi tüüpe ja väärtusi ning tüübiklasse.

Selles jaotises vaatleme kõige lihtsamaid andmemuutujadeklaratsioone, mis koosnevad võrdusmärgiga eraldatud näidisest (vasakul pool) ja avaldisest (paremal pool). Selline deklaratsioon defineerib (üldiselt võibolla koos teiste deklaratsioonidega — selliseid juhte praegu ei vaatle) muutujatele, mis vasakus pooles esinevad, väärtused. Peatüki alguses õppisime, et näidis on süntaktiline konstruktsioon etteantud väärtuse järgi muutujate väärtuse leidmiseks. Põhimõte on, et parema poole avaldise väärtuse järgi määratakse vasaku poole muutujate väärtused.

Etteantud väärtuse järgi muutujate väärtuse leidmine näidisest aga alati ei õnnestu. Väärtus võib näidisega sobituda (ingl *match*) või mitte. Kui väärtus sobitub näidisega, määrab see tüüpiliselt kõigi näidises esinevate muutujate väärtused (kuid nn laisaks sunnitud näidiste puhul ei pruugi see nii olla). Kui väärtus näidisega ei sobitu, pole muutujate väärtuse leidmine võimalik.

Kontrollimaks sobitumist, väärtustatakse avaldist ainult niipalju, et sobitumine või mittedobitumine selguks, mitte tingimata lõppvastuseni. See on Haskellis laisa väärtustamise (ingl *lazy evaluation*) tuum.

Selles jaotises vaatame muuhulgas üle kõik Haskellis peamised näidiseliigid ja õpime, milliste väärtustega nad sobituvad. Puutumata jäävad ainult mõned vähemtähtsad näidiseliigid. Näidised näevad suures osas välja nagu avaldised ja neis kehtivad samad infiksoperaatorite prioriteedid ja assotsiatiivsused kui avaldistes. Kuigi selles jaotises vaatleme näidiseid vaid kõige lihtsamas kasutuses, on näidistega opereerimise põhimõtted samad igal pool.

Esimene võimalus on, et näidis vasakul pool on lihtsalt muutuja. Näiteks deklaratsioon

```
k = 10
```

 (2)

defineerib muutuja `k` väärtuseks 10.

Ülesandeid

42. Kirjutada deklaratsioon (2) oma moodulisse. Testida seda interaktiivses interpretaatoris.
43. Lisada deklaratsioon, mis defineerib muutuja `e` väärtuseks `e`.

Muutujanäidisega sobitub mistahes väärtus, isegi `bottom`, sest avaldise muutujanäidisega sobitamisel ei väärtustata. Muidugi kui defineerida

```
k = error "VIGA"
```

ja lasta siis k väärtustada, tekib viga, kuid see ei teki näidisesobitamisel, vaid pärast seda, parema poole avaldise väärtustamisel. Hiljem vaatleme olukordi, kus on võimalik ka interaktiivse interpretaatori abil demonstreerida, et muutujaga sobitub isegi vigane väärtus.

Teine põhiline juht on näidise konstrueerimine mitmest osast, mis on omakorda näidised, kasutades konstruktorit. Kuna andmekonstruktor sisuliselt moodustab oma argumentidest andmestruktuuri, siis selle konstruktori abil ehitatud väärtuse järgi on argumentide väärtused üheselt välja loetavad, nii et selline konstruktsioon tõesti sobib näidiseks. Tüüpiline näide on deklaratsioon

$$(x, y) = (5, 0.2), \quad (3)$$

mis määrab x väärtuseks 5 ja y väärtuseks 0.2.

Ülesandeid

44. Definieerida ühe deklaratsiooniga kaks muutujat, mille väärtusteks on vastavalt Teie ees- ja perekonnanimi stringina. Anda interaktiivse interpretaatori käsurealt 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.
45. Kirjutada oma moodulisse deklaratsioon, mis defineerib muutujate q ja r väärtusteks vastavalt arvujärgendi 0.50001, 0.50002, ..., 1.49999 arvude korrutise täis- ja murdosa. Anda interpretaatori käsurealt avaldis, mille väärtuseks on korrektne eestikeelne lause, mis ütleb, mis on täisosa ja mis on murdosa.
46. Kirjutada deklaratsiooniga (3) analoogiline deklaratsioon, mille vasakus pooles on keerulisem näidis.

Kuna $:$ on konstruktor, siis on meil võimalik kasutada näidiseid kujul $p : q$. Triviaalne näide oleks deklaratsioon

$$p : ps = 1 : 3 : 4 : [], \quad (4)$$

mis annab muutujale p väärtuseks 1 ja muutujale ps väärtuseks $3 : 4 : []$, st 2-elementilise listi elementidega 3 ja 4.

Soovi korral võime rohkem elemente algusest muutujatega välja tuua, näiteks deklaratsioon

$$p : q : qs = 1 : 3 : 4 : []$$

defineerib p väärtuseks 1, q väärtuseks 3 ja qs väärtuseks listi, mille ainus element on 4.

Kasutades listide esitust elementide loeteluna, saame deklaratsiooni (4) samaväärselt kirjutada kujul

$$p : ps = [1, 3, 4].$$

Seda erisüntaksit lubab Haskell kasutada ka näidistes.

Ülesandeid

47. Modifitseerida deklaratsiooni (4) paremat poolt nii, et `ps` väärtuseks saaks tühi list. Teha seda nii erisüntaksiga kui ilma.
48. Kas saab kirjutada deklaratsiooniga (4) samaväärsse deklaratsiooni, mis kasutab listinäidist elementide loetelu kujul?

Kuna stringid on sümbolite listid, on korrektne ka näiteks deklaratsioon

```
c : cs = "Hei!",
```

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

Ka stringide erisüntaksit sümbolite joruga jutumärkide vahel saab kasutada näidistes.

Konstruktori abil moodustatud näidisega sobivad ainult sellised väärtused, mis on konstrueeritud näidises esitatud konstruktoriga väärtustest, mis sobituvad vastavate alamnäidistega. Vigased väärtused nende hulka ei kuulu.

Näiteks kui modifitseerida deklaratsiooni (4) nii, et paremale poole jääb ainult `[]`, siis parema poole väärtus pole saadav `:` abil ja seetõttu vasakul oleva näidisega ei sobitu. Muutujate `p` ja `ps` väärtust pole võimalik määrata. Kuigi selline deklaratsioon on igati korrektne ja kompileerub, annab `p` või `ps` väärtustamine täitmisaegse vea.

Samuti pole deklaratsioon

```
(0 , x1) = (log 5 , 9) (5)
```

kasutatav, kuigi ta on igati korrektne ja kompileerub. Muutuja `x1` väärtustamine nõuab kõigepealt tema deklaratsiooni parema poole väärtuse sobitamist vasaku poolega, mis ei tule välja, sest $\ln 5 \neq 0$, näidiseaga `0` aga sobitub ainult väärtus `0`.

Deklaratsiooniga (5) sarnasel kujul deklaratsioonist

```
(zzz , x2) = (error "VIGA" , 9) (6)
```

õnnestub aga `x2` väärtus `9` kätte saada, sest nüüd on esimeses komponendis muutuja, millega sobitub isegi vigane väärtus. Muutuja `zzz` väärtustamine tekitaks muidugi vea.

Ülesandeid

49. Toksida sisse deklaratsioon (5) ja vaadata järele, kuidas interaktiivne interpretaator sellisel juhul käitub, kui näidisesobitus ebaõnnestub. Jätta meelde veateate väljanägemine, et teinekord tunneks ära.

50. Toksida sisse ka deklaratsioon (6) ja kontrollida interaktiivses interpretaatoris, et kuigi `z z z` väärtustamine annab vea, õnnestub `x 2` väärtus kätte saada.

51. 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. Kontrollida oma otsust interpretaatoriga. Kui mõni otsus osutus valeks, kirjutada deklaratsioon samaväärselt ümber kujul, kus listide esitust elementide loeteluna ei kasutata, ja uurida selle kuju pealt, miks otsus vale oli.

Näidise moodustamisel konstruktoriga viitavad muutujad struktuuri omavahel lõikumatutele osadele. Näiteks näidises (x, y) viitavad x ja y paari erinevatele komponentidele, mis omavahel ei lõiku. Haskell lubab ka olukorda, kus näidise muutujate väärtustest üks on teise osa. Seda võimaldab nn *kuinäidis* (ingl *as-pattern*), mis konstrueeritakse sümboliga `@` muutujast ja suvalisest näidisest.

Näiteks võiksime täiendada deklaratsiooni (3) vasakut poolt selliselt, et lisaks muutujate x ja y väärtuse määramisele kirjutataks kogu paari väärtus muutujasse z :

$$z@ (x, y) = (5, 0.2). \tag{7}$$

Niisiis on `@` sisuliselt defineeriv võrdus nagu `=`, ainult et tema mõlemal pool on näidised. Kuinäidise konstrueerimisel tuleb arvestada, et `@` seob tugevamini kõigist infiksoperaatoritest ja isegi funktsioonirakendamisest (piltlikult öeldes on `@` prioriteediga 11; niisama piltlikult võib näiteks öelda, et paarikonstruktor on prioriteediga -1 , sest paar peab alati sulgudes olema).

Ülesandeid

52. Vaatleme deklaratsiooni

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

Millised väärtused omandavad selle deklaratsiooni tulemusel muutujad `t`, `u`, `ts` ja `us`? Mõelda enne ja siis kontrollida interaktiivse interpretaatoriga.

53. Kirjutada võimalikult lühike deklaratsioon, mis defineerib muutuja `eee` väärtuseks arvu `ee` ja muutuja `eeeList` väärtuseks listi, mille ainsaks elemendiks on seesama väärtus.

54. Kirjutada üks deklaratsioon, mis defineerib muutujate c_1, c_2, c_3 väärtuseks vastavalt 1, 2, 3 ning muutuja d väärtuseks (2, 3). Ükski avaldis paremal pool ärgu kordugu.

Veel üks näidiseliik on jokker (ingl *wildcard*), mis ei anna väärtust ühelegi muutujale. Muus osas on jokkeriga opereerimine samasugune kui muutujaga, lihtsalt väärtus, mis temaga sobitatakse, unustatakse ära. Haskellis tähistab jokkerit üksik alakriips (st alakriips, mis ei kuulu identifikaatori koosseisu). 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 (7), 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 (7) y asendada jokkeriga:

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

Mittekasutatavate muutujate asendamine jokkeriga teeb programmi loetavamaks, sest säästab lugejat nende muutujate tähenduse meelestpidamisest.

Ülesandeid

55. 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.

Öeldakse, et näidis on laisk (ingl *irrefutable*), kui iga väärtus sobitub temaga. Vastasel korral öeldakse, et näidis on agar (ingl *refutable*). Samaväärselt võiks öelda, et näidis on laisk, kui vigane väärtus bottom sobitub temaga, ja agar, kui bottom ei sobitu temaga. Masina töö aspektist vaadates on näidis agar, kui avaldise väärtuse sobituvuse kontroll nõuab avaldise väärtustamist vähemalt välimise konstruktori ilmumiseni, ja laisk, kui see kontroll ei nõua avaldise väärtustamist üldse.

Seniõpitud näidistest on muutuja ja jokker laisad ning kõik konstruktoriga ehitatud näidised agarad. Kuinäidised on laisad või agarad vastavalt sellele, kas $@$ parem näidis on laisk või agar.

Viimane näidiseliik, mida vaatleme, on laisaks sunnitud näidis. Igast nädisest saab moodustada laisa näidise, lisades tema ette tilde (ülejäanud osa nädisest tuleb siis tüüpiliselt sulgudesse panna).

Näiteks kui deklaratsioonis (5) sundida näidis 0 laisaks, kirjutades tema ette tilde, siis väärtus $\ln 5$ sobitub temaga kõigest hoolimata ning $x1$ väärtustamisel viga ei teki.

Laisaks sundimine seega võimaldab mõnikord ennetada täitmisaegset viga, kui ta tekib meie vajaduste suhtes n -õ perifeersetel põhjustel nagu siin, kus $x1$ väärtustamisega kaasneb millegi muu (paari teise komponendi) sobitamine nädisega, mis isenesest ei puutu asjasse.

Laisaks sundimine muudab näidisesobitaja selle näidise suhtes n-ö pimedaks, ta näib näidisesobitajale kui jokker ja temas esinevad muutujad jäävad näidisesobitamisel väärtustega sidumata. Juhul, kui edasise arvutuse käigus mõnd neist muutujatest siiski vajatakse, avatakse näidis ja püütakse väärtus sobitada tagantjärele. Laisaks sundimise tulemus ongi põhimõtteliselt näidisesobitamise edasilükkamine ajaks, kui selgub, et see on tõesti vajalik. Kui arvutusprotsess lõpeb enne, st näidisesobitamine ei olnud vajalik, siis see jääbki tegemata.

Laisaks sundimisega peab olema ettevaatlik, sest kui väärtus, mida sellise näidisega sobitatakse, oma struktuurilt näidisele ei vasta ja lastakse väärtustada mõni selle näidise sees olev muutuja, siis temale väärtuse leidmine pole võimalik ja tekib täitmisaegne viga. Näiteks deklareerides

$$\sim(p : ps) = [],$$

tekib p või ps väärtustamisel täitmisaegne viga sellest hoolimata, et väärtus näidisega sobitub — kuna parema poole väärtuse struktuur näidise struktuurile ei vasta, pole muutujatele väärtusi kuskilt võtta.

Ülesandeid

56. Kontrollida analoogselt muutujanäidisega, et laisaks sunnitud näidisega sobitub ka vigane väärtus.

57. Olgu meil deklaratsioon

$$\sim(x, (a, b)) = (1, \text{error "VIGA"}).$$

Selgitada, miks muutuja x väärtustamine lõpeb veateatega, kuigi vasakul esineb tilde. Tõsta antud deklaratsioonis 1 sümbol ümber nii, et x väärtustamine annaks väärtuseks 1.

Näidiseid sisaldavad avaldised, keerulisemad andmedeklaratsioonid

Alustame avaldistest, mille väärtus on funktsioon. Niisugused avaldised algavad võtmesümboliga \backslash , millele järgnevad sümbolikombinatsiooniga \rightarrow eraldatud näidis ja avaldis. Noole moodi kombinatsioon \rightarrow nagu varem võrdusmärkki jagab kirjutise vasakuks ja paremaks pooleks. Sümbolit \backslash loetakse “lambda”. Kogu selle n-ö lambdaavaldise väärtuseks on funktsioon, mis igal oma argumendil annab tulemuseks parema poole avaldise väärtuse, kus muutujate väärtused tuletatakse argumendi sobitamisel vasaku poole näidisega.

Näiteks avaldis $\backslash x \rightarrow x * x$ väärtuseks on ruutfunktsioon. Kui lastakse väärtustada avaldis kujul $(\backslash x \rightarrow x * x) a$ mingi a korral, siis sobitatakse a vasaku poole näidisega ehk muutujaga x , mis defineerib muutuja x väärtuse võrdseks a väärtusega, ning antakse tulemuseks parem pool, seega $a * a$.

Lambdaavaldisel vasakus pooles esinevad muutujad on lokaalsed, nähtavad ainult selle lambdaavaldisel piires. Näiteks x eelmise lõigu näiteavaldises on lokaalne muutuja, tema nähtavus piirduv ainult selle lambdaavaldisega. Põhimõtteliselt tohivad lokaalsete muutujate nimed lange- da kokku globaalsete muutujate nimedega, sellisel puhul varjutatakse vastavad globaalsed muu- tujad lokaalse muutuja nähtavuspiirkonnas, nad pole seal kasutatavad.

Avaldisel $\lambda (x, _) \rightarrow x$ väärtuseks on paari projektsioon esimesele argumendile, st sama mis prelüüdis defineeritud muutujal fst . Jokkeri asemel võiks olla ka mõni x -st erinev muutuja, aga kuna seda muutujat kuskil ei kasutata, on jokker sobivam.

Niisiis lambdaavaldisel puhul on liikumise järjekord vastupidine eelmises jaotises õpitud dekla- ratsioonidega võrreldes: kui seal liikus väärtus avaldisest näidisesse ja määras lõppkokkuvõttes näidise muutujate väärtused, siis siin liiguvad muutujate väärtused näidisest avaldisse ja määra- vad nii funktsiooni väärtuse, näidise muutujate väärtused aga määrab argument.

Ülesandeid

58. Testida ruutfunktsiooni interaktiivse interpretaatori käsurealt.
59. Kuidas testida, et $\lambda (x, _) \rightarrow x$ ja fst on sama väärtusega?
60. Kirjutada lambdaavaldis, mille väärtus on funktsioon, mis võtab argumendiks arvupaari ja annab väärtuseks komponentide vahe absoluutväärtuse.
61. Kirjutada lambdaavaldis, mille väärtus on sama mis prelüüdis defineeritud muutujal (a) `head`, (b) `tail` (seejuures neid muutujaid kasutamata). Kus võimalik, kasutada jokkerit.
62. Kirjutada võimalikult lühike ja võimalikult vähe muutujaid kasutav avaldis, mille väärtus on funktsioon, mis võtab argumendiks paari ja annab välja kolmiku, mille esimesed kaks komponenti võrduvad argumentpaari esimese komponendiga ja viimane komponent on argumentpaar ise.

Kui meil on ruutfunktsiooni vaja tihti kasutada, tahaksime kindlasti tema jaoks spetsiaalse muu- tuja defineerida, nagu näiteks muutuja fst väärtuseks on prelüüdis defineeritud teatav projekt- sioonifunktsioon. See on täiesti tehtav eelmises jaotises vaadeldud tüüpi deklaratsiooniga:

```
sqr = \ x -> x * x.
```

Samas on funktsioonide defineerimiseks olemas ka eraldi deklaratsiooniliik. Erinevalt siiani vaa- delduist on niisuguses deklaratsioonis vasakul rohkem kui üks näidis. Ruutfunktsiooni saaksime sellega ümber kirjutada kujul

```
sqr x
  = x * x
```

Niisiis lambda on kadunud, näidis lambda alt on liikunud vasakusse poolde ning avaldis lambda alt moodustab üksi terve parema poole. Niisugune deklaratsioon defineerib muutuja `sqr` väärtuseks funktsiooni, ilma et temas leiduks selle väärtusega avaldist.

Niisugune viis funktsioone muutujatele väärtuseks anda on pisut mugavam kui eelmine ja enamasti eelistataksegi seda.

Ülesandeid

63. Leida moodulist Prelude funktsioonide `fst` ja `snd` ning `head` ja `tail` definitsioonid ja saada neist aru.
64. Defineerida uue meetodiga muutuja `diff`, mille väärtus on kirjeldatud ülesandes 60.
65. Võimalikult lühikese ja võimalikult vähe muutujaid kasutava deklaratsiooniga defineerida muutuja `imelik`, mille väärtus on kirjeldatud ülesandes 62.

Kuna järgnevalt õpitavad avaldised on liiga pikad, et neid interaktiivse interpretaatori käsurealt sisestada, siis käsitleme neid ainult deklaratsioonide koosseisus.

Mitmerealised avaldised ja deklaratsioonid võivad halva paigutuse korral kompilaatori segadusse ajada ja ta võib süntaksiveast teatada, kuigi Teil võisid olla parimad kavatsused. Paigutades koodi nii, nagu on toodud näidetes, selliseid probleeme ei teki.

Kõige üldisem kontroll sisaldav konstruktsioon on **valikuavaldis**. Otsustus tehakse näidiseso-bitamiste abil. Näiteks järgmine triviaalne funktsioon teisendab tõeväärtuse vastavaks arvuks:

```
tõeväärtusArvuks x
= case x of
  True      -> 1
  _         -> 0
```

 (8)

Järgmine funktsioon võtab argumendiks listi ja annab väärtuseks eestikeelse teksti, mis ütleb, kas listis oli elemente null, üks või mitu:

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

 (9)

Järgmine funktsioon võtab samuti argumendiks listi ja kui see on mittetühi, siis annab väärtuseks listi, mis on argumendist saadud esimese elemendi ümbertõstmisel listi lõppu, tühjal listil aga annab tühja listi:

```

üksLõppu xs
  = case xs of
      z : zs      -> zs ++ [z]
    -
      -          -> []

```

(10)

Süntaksi poolest on valikuavaldise igale näidisele vastav juht nagu tavalise definitsiooni parem pool, milles võrdusmärkide asemel on nooled. Valikuavaldise väärtus võrdub üldjuhul esimese sellise parema poole avaldise väärtusega, millele vastava vasaku poole näidisega päiseavaldise (st võtmesõnade **case** ja **of** vahelise avaldise) väärtus sobitub. Kui sellist näidist ei leidu, on valikuavaldise väärtus bottom. Erandiks on olukord, kus päiseavaldise väärtus on bottom ja esimene näidis on agar — sel juhul on kogu valikuavaldise väärtus bottom. Asi on selles, et bottomit pole alati võimalik tuvastada (nimelt kui põhjuseks on lõpmatu arvutus) ja seega pole võimalik nõuda, et bottomit järjest mitme agara näidisega sobitataks.

Deklaratsioonis (8) oleva valikuavaldise puhul käib täitmine järgnevalt. Muutuja x väärtustatakse niikaugele, et saaks kontrollida sobituvust esimese näidisega `True`. Kui x väärtustamine viib väärtuseni `True`, siis sobitamine õnnestub ja kogu avaldise väärtuseks saab 1. Kui x väärtustamine viib väärtuseni `False`, siis sobitamine ei õnnestu ja minnakse järgmise näidise juurde. Siin on järgmine näidis jokker, millega sobitub triviaalselt iga väärtus, muuhulgas `False`, seega kogu avaldise väärtuseks saab 0. On veel üks võimalus — x väärtus on bottom (rohkem võimalusi pole, sest x peab olema tõeväärtustüüpi, muidu poleks antud avaldis korrektné). Sel juhul võtab esimene agar näidis `True` ta rajalt maha ja kogu avaldise väärtus on samuti bottom.

Funktsiooni `primLoenda` defineerimine deklaratsiooniga (9) on tegelikult väga ebaõnnestunud, sest esimese näidisega sobitumise kontrolliks tuleb argumentlisti pikkus välja arvutada (sest enne ei ilmu konstruktorit), see on aga lineaarse keerukusega 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` jääbki mõtlema või lõpetab töö veateatega, samas kui kahe esimese elemendi leidmise järel võiks vastuse “Palju” välja anda.

Ülesandeid

66. Kirjutada täisarve tõeväärtuseks teisendav funktsioon `intValue`, mis 0 puhul annab `False` ja kõigi ülejäänud arvude puhul `True`.
67. Kirjutada eelmise ülesande funktsiooni modifikatsioon `intValue'`, mis lõpetab töö Teie etteantud veateatega, kui argument pole 0 ega 1.

68. Kirjutada funktsioon `primLoenda` uus definitsioon, mis töötab listi pikkuse suhtes konstantse keerukusega. Funktsioon peab vastuse välja andma ka lõpmatul listil ja osalisel listil, milles on vähemalt üks element.
69. Kirjutada funktsioon `esiPaar`, mis võtab argumendiks listi ning, kui selles on vähemalt 2 elementi, annab väärtuseks paari kahest esimesest elemendist, vastasel korral lõpetab oma etteantud veateatega. Funktsioon peab töötama konstantse keerukusega listi pikkuse suhtes.
70. Kirjutada funktsioon `esiSuureks`, mis võtab argumendiks stringi ja annab väärtuseks stringi, mille saab argumentstringist esimese tähe suureks muutmise teel, kui string on mitetühi, ja argumentstringi enda vastasel korral.
71. Oletame, et funktsiooni `primLoenda`, mis on defineeritud ülesandes 68, läheb vaja ainult stringide jaoks. Kas on võimalik kirjutada alternatiivne definitsioon, mis sisaldab näidistena vaid stringinäidiseid?

Kui funktsiooni definitsiooni parem pool koosneb valikuavaldisest, mille päiseavaldis langeb kokku argumendinäidisega ja argumendinäidise muutujad mujal ei esine, nagu see on deklatsioonid (8) ja (10), saab Haskellis seda funktsiooni samaväärselt defineerida ka ilma valikuavaldist sisse toomata, kirjutades iga valiku jaoks eraldi deklaratsiooni. Kooditükkide liikumine paremast poolest vasakusse on analoogiline ülal nähtud näidise liikumisega lambda alt vasakusse poole.

Näiteks funktsioonid `tõeväärtusArvust` ja `üksLõppu` oleksid sellisel viisil defineeritud kumbki kahe deklaratsiooniga:

```
tõeväärtusArvuks True
  = 1
tõeväärtusArvuks _
  = 0

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

Ülesandeid

72. Defineerida uue meetodiga funktsioon `intValue'` ülesandest 67 ja `primLoenda` ülesandest 68.

Taolist funktsiooni nagu `tõeväärtusArvuks` ei defineerita praktikas aga ühelgi seni käsitletud viisil, sest Haskellis on tõeväärtusnäidistega sobitamiseks eraldi süntaktiline konstruktsioon

— tingimusavaldis. Selle abil saab sama funktsiooni defineerida vähem kohmakalt deklaratsiooniga

```
tõeväärtusArvuks x
  = if x then 1 else 0
```

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 *else*-haru ning need peavad olema sama tüüpi. Teiseks pange tähele, et tegemist on tõepoolest avaldisega, mitte mingi deklaratsiooni ega käsuga. C-s ja Javas on tema analoogiks konstruktsioon $\langle \text{tingimus} \rangle ? \langle \text{avaldis} \rangle : \langle \text{avaldis} \rangle$, mitte *if*-konstruktsioon.

Tihti, eriti kui tingimusi on rohkem kui üks, on tingimusavaldiste asemel mugavam kasutada valvureid (ingl *guard*). Iga valvur on tõeväärtustüüpi avaldis ehk tingimus, ta asub püstkriipsu järel ning talle järgneb vahemärgiga eraldatult avaldis, mis tuleb valida selle tingimuse täidetuse korral. Valvurid oleks sobilik rajastada üksteise alla.

Oletame, et on vaja kirjutada funktsioon `arvuKlass`, mis arvulise argumendi korral annab väärtuseks teksti “Negatiivne”, “Null”, “Üks”, kui argument on vastavalt negatiivne, 0 või 1, ja “Suur” ülejäänud juhtudel. Siis tingimusavaldis läheks suhteliselt kohmakaks, sest kahe haru asemel on siin neli, tingimusavaldisi tuleks üksteise sisse panna. Ka valikuavaldisega pole lood paremad. Erinevalt valikuavaldisest deklaratsioonis (9) on siin kaks “suurt juhtu”, mis hõlmavad paljusid arve, pole võimalik kumbagi eraldi ühegi näidisega parajalt katta, peaksime jälle ühe hargnemiskonstruktsiooni teise sisse panema. Valvurid annavad meile võimaluse neli juhtu ühel tasemel kirjeldada:

```
arvuKlass x
  | x < 0
  = "Negatiivne"
  | x == 0
  = "Null"
  | x == 1
  = "Üks"
  | x > 1
  = "Suur"
(11)
```

Kui valikuavaldise puhul toimus ühe väärtuse sobitamine järjest paljude näidistega, siis siin toimub paljude väärtuste (tingimuste) sobitamine järjest ühe näidisega (`True`).

Järjekord on ka valvurite puhul ülalt alla ja valitakse parem pool, millele vastav valvur esimesena `True` annab. See tähendab, et kui viimane valvur ammendab kõik eelmistest ülejäävad juhud, võib tema asendada tõeväärtusega `True` — mõlemal juhul väärtustub valvur tõeväärtuseks `True` — ja see on efektiivsuse mõttes kasulik, kuna üks arvutus jääb ära. Et `True` viimase valvurina võib koodi loetavust halvendada, on prelüüdis defineeritud muutuja `otherwise`,

mille väärtus on True ja mis on mõeldud viimase ammendava valvurina kasutamiseks. Näiteks deklaratsioon (11) on samaväärselt ümberkirjutatuna

```
arvuKlass x
| x < 0
  = "Negatiivne"
| x == 0
  = "Null"
| x == 1
  = "Üks"
| otherwise
  = "Suur"
```

Ülesandeid

73. Leida moodulis Prelude koht, kus `otherwise` defineeritakse.

Valvureid saab samamoodi kasutada ka valikuavaldise all kohe näidise järel. Näiteks ütleme, et list on tore, kui tal on olemas positiivne esimene element, ja käib kah, kui esimene element on null. Kui listi esimene element on negatiivne või on list tühi, siis ta ei sobi meile. Järgnev funktsioon paneb paika listi klassi sellises mõttes:

```
listiKlass xs
= case xs of
  z : _
    | z > 0
      -> ":( "
    | z == 0
      -> ":( | "
  _
    -> ":( "
```

See näide demonstreerib üht põhimõttelist erinevust valikuavaldise ja valvurikonstruktsiooniga käitumisel. Paneme tähele, et näidise `z : _` all valvurid ei ammenda kõiki võimalikke juhte. Kui juhtub, et ükski valvur ei sobi, siis pöörduakse tagasi ühe taseme võrra väljapoole, valides näidiste nimekirjast järgmise. Valvurite nimekirja lõppemine iseenesest ei tekita täitmisaegset viga. Kui aga näidised otsa saavad, siis enam väljapoole tagasi ei pöördata ja antakse viga nagu juba ülalpool seletatud.

Funktsiooni `listiKlass` samaväärne definitsioon mitme deklaratsiooniga on

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

On võimalik ka samaväärne definitsioon ühe deklaratsiooniga, kasutades laisaks sundimist:

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

(12)

Kuna argumentis on laisk näidis, sobitub iga argumentlist triviaalselt ja minnakse funktsiooni keha juurde. Kui argumentlist on tühi, väärtustub esimese valvuri disjunktsiooni vasak pool ja seega ka kogu valvur tõeseks ilma disjunktsiooni teist poolt uurimata ning väärtuseks saadakse “:(”. Tänu `||` laiskusele niisugune definitsioon töötab: kui `||` nõuaks ka oma parema argumenti väärtustamist, järgneks siin viga, kuna muutujat `z` pole võimalik määrata. Kui list on mittetühi, siis esimese valvuri disjunktsiooni vasak pool väärtustub vääraks ja seega väärtustatakse ka disjunktsiooni parem pool. On vaja muutujat `z`, kuid kuna nüüd on list mittetühi, õnnestub list jooksvalt sobitada näidisega `z : _` ja `z` määrata. Edasi on juba selge — käitatakse vastavalt `z` suurusele.

Ülesandeid

74. Kas deklaratsioonis (12) disjunktsiooni poolte vahetamisel saadakse samaväärne deklaratsioon?

Lokaalsete deklaratsioonidega konstruktsioonid

Avaldistes, mis muidu läheksid liiga pikaks, saab osade väärtusi omistada lokaalsetele muutujatele, kasutades *let*-avaldist nagu 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
```

Lokaalseteks deklaratsioonideks võivad olla igasugused andmedeklaratsioonid. *Let*-avaldise väärtus võrdub võtmesõna **in** järel oleva avaldise väärtusega. Lokaalsete muutujate skoop hõlmab parajasti kogu selle *let*-avaldise.

Juhul, kui arvutusi sisaldavad avaldised kipuvad väärtustuma korduvalt, tasub lokaalse deklaratsiooniga nende avaldiste väärtused muutujatega siduda. See võimaldab kilplasilikest arvutustest vabaneda, sest muutujad väärtustatakse Haskellis ülimalt üks kord.

Ülesandeid

75. Kirjutada funktsioon `summanali`, 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.
76. Kirjutada funktsioon `täisMurdNali`, mis võtab argumendiks reaalmurdarvu a ja annab väärtuseks arvu $\lfloor a \rfloor^{(a)} \cdot \langle a \rangle$, kus $\lfloor x \rfloor$ ja $\langle x \rangle$ tähistavad vastavalt arvu x täis- ja murdosa.

Kui mõnd avaldist kasutatakse korduvalt mitmes valvuris või mitmele valvurile järgnevates avaldistes, siis *let*-avaldist pole võimalik kasutada, kuna valvurid koos paremate pooltega ei moodusta kokku avaldist. Selleks puhuks on olemas *where*-konstruktsioon, mis võimaldab defineerida üle kogu parema poole (mis hõlmab kõiki valvureid ja nende vastavaid avaldise) nähtavaid lokaalseid muutujaid. *Where*-konstruktsioon kirjutatakse kogu parema poole lõppu (st üldjuhul alla) võtmesõna **where** järele.

Järgnev näitefunktsioon `veerand` võtab argumendiks ujukomaarvu ja, interpreteerides seda

nurga suurusena, tuvastab, millises veerandis on vastav nurk:

```
veerand x
| sin > 0 && cos > 0
  = "I" ++ v
| sin > 0 && cos < 0
  = "II" ++ v
| sin < 0 && cos < 0
  = "III" ++ v
| sin < 0 && cos > 0 .
  = "IV" ++ v
| otherwise
  = "vahepeal"
where
sin = Prelude.sin x
cos = Prelude.cos x
v = " veerandis"
```

See näide demonstreerib ka, et lokaalsed muutujanimed võivad globaalseid ja isegi standardteegis defineeritud muutujaid varjata. Antud olukorras defineeritakse lokaalselt üle muutujanimed `sin` ja `cos`. Kuna nad defineeritakse prelüüdi vastavanimeliste muutujate kaudu, tuleb originaalsetele prelüüdi muutujatele viitamisel kasutada kvalifitseerimist, st moodulinime lisamist nime ette.

Ülesandeid

77. Kirjutada funktsioon `sümbolid`, 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` või `Prelude`.

Viimane süntaktiline konstruktsioon, mida siin jaotises vaatame, kannab **komprehensioonsüntaksi** (ingl *comprehension syntax*) nime. See on analoogne matemaatikast tuntud viisile esitada hulki teatud kitsenduste kaudu nagu näiteks kujus $\{x^2 \mid 1 \leq x < 10\}$, mis märgib hulka kõigist 10-st väiksemate positiivsete täisarvude ruutudest. Haskellis saab vastava listi koostada avaldisega `[x * x | x <- [1 .. 9]]`. Muutuja `x` läbib järjekorras kõik listi `[1 .. 9]` elemendid ja tulemuslist moodustub vastavatest avaldise `x * x` väärtustest samas järjekorras.

Komprehensioonsüntaksis võib muutujatele seada ka lisatingimusi. Näiteks kui meid arvu 5 ruut ei huvita, siis võime selle ruutude listist välja jätta, kirjutades avaldise

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

 (13)

Samuti võib korraga kasutada mitut muutujat. Näiteks avaldise

$$[(x, y) \mid x \leftarrow [1 \dots 9], y \leftarrow [1 \dots 9], x \leq y]. \quad (14)$$

väärtus on list arvupaaridest, mille mõlemad komponendid on positiivsed ja 10-st väiksemad ning teine komponent pole esimesest väiksem. Pangem tähele paaride järjekorda, sest see kajastab muutujate väärtuste muutumise järjekorda, mis on alati sama: parempoolsemad muutujad kerivad kiiremini.

Komprehensioonsüntaksi elemente kujul $p \leftarrow a$ nimetatakse generaatoriteks (ingl *generator*), lisatingimusi aga jällegi valvuriteks (ingl *guard*). Generaatori vasakuks pooleks on näidis, paremaks listitüüpi avaldis. Valvur on suvaline tingimus, st tõeväärtustüüpi avaldis.

Kui generaatoris $p \leftarrow a$ juhtub, et listi \bar{a} mõni element näidisega p ei sobitu, jäetakse ta lihtsalt vahele, viga üldjuhul ei teki (tekib ainult siis, kui element on bottom). Näiteks järgnevalt defineeritud funktsioon `pead` võtab argumentiks listide listi ja annab väärtuseks selle listi kõigi mittetühjade elementide esimesed elemendid:

$$\begin{aligned} \text{pead } xss \\ = [x \mid x : _ \leftarrow xss] \end{aligned} \quad (15)$$

Kasutades deklaratsioonis (15) defineeritud funktsiooni `pead`, saame suvalise listide listi l mittetühjade elementide arvu leida muidugi avaldisega `length (pead a)`, kus $\bar{a} = l$. Kui aga ainult nende arv huvitabki, siis võime `pead` asemel defineerida otse seda arvutava funktsiooni näiteks kujul

$$\begin{aligned} \text{mittetühjadeArv } xss \\ = \text{sum } [1 \mid _ : _ \leftarrow xss] \end{aligned} \quad (16)$$

Näidises on ka listi pea asendunud jokkeriga, kuna summa arvutamisel ei lähe tema väärtust vaja. Näidis ei seo sobitamisel ühtki muutujat, kuid konstruktsioon kooloniga ütleb jätkuvalt, et sobituvad vaid mittetühjad listid. Komprehensioonavaldise väärtus on list, milles on üks element 1 argumentlisti iga mittetühja elemendi kohta. Nende summeerimisel ilmselt saame listi elementide arvu nagu vaja.

Ülesandeid

78. Kirjutada komprehensioonsüntaksiga avaldis, mille väärtus on sama mis avaldisel (13), kuid ei sisalda valvureid.
79. Kirjutada komprehensioonsüntaksiga avaldis, mille väärtus on sama mis avaldisel (14), kuid ei sisalda valvureid.
80. Kirjutada avaldis, mille väärtus on kolmas täisruut, mis on suurem kui 100000.

81. Kui deklaratsioonis (15) sundida generaatori vasak pool tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust?
82. Kui deklaratsioonis (16) sundida generaatori vasak pool tildega laisaks, kas ja kuidas see muudab defineeritava funktsiooni väärtust?
83. Kas listide listi mittetühjade elementide arvu leidmise funktsiooni saaks samaväärselt defineerida ka deklaratsiooniga

```
mittetühjadeArv xss
= length [error "Jube viga" | _ : _ <- xss]?
```

Komprehensioonsüntaksi kasutamisel tuleb valvurid panna nii varakult kui võimalik, st iga valvur panna kohe nende generaatorite järele, mis defineerivad valvuris esinevad muutujad. Kui valvuri ees on generaatoreid, mis defineerivad muutujaid, millest valvur ei sõltu, tekitab see tarbetu töökulu arvutuse käigus, sest valvuri väärtust kontrollitakse selle lisamuutuja kõigi väärtuste korral uuesti. Näiteks avaldise

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

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

Lisaks generaatoritele ja valvuritele võib komprehensioonavaldise sees esineda ka lokaalseid andmedeklaratsioone. Lokaalsete andmedeklaratsioonide plokid algavad võtmesõnaga `let` ja nad eraldatakse komadega nagu generaatorid ja valvuridki.

Funktsioonid funktsiooni väärtusena ja argumendina

Haskellis on kõik funktsioonid formaalselt üheargumendilised — töötavad mingist kindlast ühest tüübist A kindlasse tüüpi B . Mitmeargumendiliste funktsioonide asemel peab niisiis Haskellis üheargumendilistega läbi ajama. Selleks on kaks standardset võimalust.

Esimene on kasutada funktsioone, mille argumenditüüp on korteežitüüp. Niisugused funktsioonid võtavad argumendiks korteeži, mis ju sisaldab endas mitut eraldi väärtust — nii sõltubki funktsiooni väärtus sisuliselt mitmest parameetrist. Kuna korteežisüntaks Haskellis on komadega eraldatud komponendid ümarsulgudes, on funktsiooni rakendamine korteežile Haskellis äravahetamiseni sarnane mitmeargumendilise funktsiooni rakendamisele standardsetes imperatiivsetes keeltes. Seega võib kinnitada, et üheargumendilisuse nõue on puhtformaalne kitsendus, mis keelt sisuliste väljendusvõimaluste poolest sugugi vaesemaks ei tee.

Funktsioonid `fst` ja `snd` on triviaalsed näited kaheargumendilisi funktsioone kirjeldatud mõttes üheargumendiliselt lavastavatest funktsioonidest. Nad annavad lihtsalt emma-kumma oma argumentidest väärtusena välja ja teise viskavad ära. Pisut sisukam on ülesannetes 60 ja 64 kirjutatud vahe absoluutväärtust arvutav funktsioon.

Haskellis lisandub siiski üks miinusmoment korteežide kasutamise juures. Kuna korteež kujutab endast andmestruktuuri, mitte lihtsalt üksikute parameetrite loetelu, siis korteežide kasutamise puhul kulub mingil määral ressursi nende andmestruktuuride tekitamiseks ning nende sobitamiseks näidistega. See on hind, mida Haskellis korteežidega programmeerija maksab tavaliste imperatiivsete keelte mitmeargumendiliste funktsioonide kasutamisega võrreldes.

Seetõttu kasutatakse Haskellis enamasti teist võimalust mitmeargumendiliste funktsioonide lahvastamiseks: kasutada funktsioone, mille väärtustüüp on funktsioonitüüp. Paljuargumendilise funktsiooni asemel kirjutatakse üheargumendiline funktsioon, mis võtab vaid esimese neist argumentidest ja annab tulemuseks funktsiooni, mis võtab argumendiks järgmise argumendi jne, niikaua kui argumente on — lõpuks antakse välja vajaliku funktsiooni väärtus kogutud argumentidel. Vahe imperatiivsete keelte paljuargumendilise funktsiooniga on niisiis selles, et see Haskellile tüüpiline konstruktsioon võtab argumente ükshaaval, mitte ühekorraga.

Olgu f korteežiargumendiga funktsioon. Siis funktsiooni, mis saadakse funktsioonist f äsjakirjeldatud skeemi järgi, nimetatakse funktsiooni f *curried*-kujuks. Funktsioone, mis on mingi funktsiooni *curried*-kujuks, nimetatakse *curried*-kujul funktsioonideks.

Näiteks funktsioonile $\backslash (x, _) \rightarrow x$ vastav uutmoodi funktsioon on

$$\backslash x \rightarrow \backslash _ \rightarrow x.$$

Ta võtab argumendiks ühe objekti ning annab väärtuseks funktsiooni, mis võtab omakorda teise objekti ning annab esimese objekti väärtuseks. See funktsioon on tegelikult prelüüdis defineeritud ja kannab nime `const`. Niisiis funktsioon `const` on funktsiooni `fst` *curried*-kuju.

Curried-kujul funktsiooni, mis arvutab kahe arvu aritmeetilist keskmist, kirjeldaks avaldis

$$\backslash a \rightarrow \backslash b \rightarrow (a + b) / 2.$$

Sama kolme argumendi korral oleks

$$\backslash a \rightarrow \backslash b \rightarrow \backslash c \rightarrow (a + b + c) / 3.$$

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 $\backslash a b \rightarrow (a + b) / 2$ ja $\backslash a b c \rightarrow (a + b + c) / 3$. Selle erisüntaksiga lisandub nõue, et sama lambda alla kogutud näidiste reas ei tohi ükski muutuja esineda korduvalt — seni kehtis see nõue vaid igas näidises eraldi.

Curried-kujul funktsioonide rakendamine argumentidele ei nõua mingeid lisateadmisi. Näiteks funktsiooni `const` rakendamiseks argumendile 5 kirjutame ikka järjest `const 5`. Tulemuseks on siin funktsioon, seega saame veel korra rakendada — näiteks argumendile 0.2:

`(const 5) 0.2`. Vastavalt `const` definitsioonile on viimase avaldise väärtuseks 5. Funktsioonid kujul `const x` argumenti ei loe — on laisad —, seega on sama väärtus ka näiteks avaldisel `(const 5) (error "E")`.

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.

Ülesandeid

84. Kirjutada ja testida interaktiivse interpretaatori käsurealt funktsiooni `snd` *curried*-kuju.
85. Kirjutada ja testida interaktiivse interpretaatori käsurealt ülesandes 64 kirjutatud funktsiooni *curried*-kuju.

Standardteegis on lisaks juba tuttavale funktsioonile `const` defineeritud veel tohutult palju funktsioone *curried*-kujul. Valdav enamik mitmeparametrisi arvutusskeeme on realiseeritud *curried*-kujul funktsioonidena.

Arvulistest funktsioonidest on *curried*-kujul `div` ja `mod`, mis võtavad ükshaaval argumentidena kaks arvu ja annavad väärtuseks vastavalt täisarvulise jagatise ja jäägi esimese arvu jagamisel teisega. Näiteks avaldised `div 10 7` ja `mod 10 7` annavad väärtuseks vastavalt 1 ja 3, sest 10 jagamisel 7-ga on jagatis 1 ja tekib jääk 3.

Curried-kujul funktsioon `subtract` annab arvulisel argumendil väärtuseks funktsiooni, mis oma argumendist lahutab selle arvu. Näiteks `subtract 2` väärtus on funktsioon, mis oma argumendist lahutab arvu 2, nii et näiteks `subtract 2 3` väärtus on 1. Funktsiooni `subtract` ühele argumendile `a` rakendades saab konstrueerida asendaja puuduvale sektsioonile kujul `(- a)`.

Veel võib kasulik olla *curried*-kujul funktsioon `compare`, mis võtab ükshaaval kaks argumenti järjestatud väärtustega tüübist ja annab tulemuseks nende võrdluse tulemuse tüübist `Ordering`. Võimalikud väärtused selles tüübis on konstruktorid `LT`, `EQ` ja `GT`, mis tähendavad `compare a b` tulemusena vastavalt, et `a` väärtus on väiksem `b` väärtusest, `a` väärtus on võrdne `b` väärtusega ja `a` väärtus on suurem `b` väärtusest.

Listifunktsioonidest võiks näiteks tuua funktsioonid `take` ja `drop`. Mõlemad võtavad argumendiks lühikese täisarvu n (st objekti tüübist `Int`) ja annavad välja funktsiooni, mis võtab argumendiks mingi listi l . Tulemuseks on esimesel juhul 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, teisel juhul aga ülejäänud elementidest. Näiteks `take 2 [5, 6, 7]` väärtuseks on 2-elementiline list elementidega 5, 6 ja `drop 2 [5, 6, 7]` väärtuseks on 1-elementiline list elemendiga 7.

Samuti *curried*-kujul funktsioon `elem` võtab üksikargumentideks objekti ja sama tüüpi objektide listi ning annab väärtuseks tõeväärtuse vastavalt sellele, kas objekt esineb listis või teda ei leitud seal — kui list on lõpmatu ja objekt seal ei esine, tekib lõpmatu arvutus ilma väljundita. Näiteks

```
elem 2 [1, 3] ==>* False,  
elem 2 [1 .. ] ==>* True,  
elem (-2) [1 .. ] jääb lõpmatusse tsükklisse, väärtus on bottom.
```

Laiemalt vaadates on aga kõik Haskellis infiksoperaatorid tegelikult *curried*-kujul funktsioonid. Näiteks muutuja `+` ja avaldis `\ a b -> a + b` on sama väärtusega. Veel enamgi — kõiki Haskellis identifikaatoreid, mille väärtuseks on *curried*-kujul funktsioon, saab kasutada infiksselt. Infiksse ja mitteinfiksse kasutuse jaoks on eraldi süntaks. Nagu juba õppisime, tuleb loendi (1) sümbolitest koosnevaid identifikaatoreid vaikumisi kasutada infiksselt, tähelisi identifikaatoreid aga vaikumisi infiksselt kasutada ei saa. Loendi (1) sümbolitest koosneva identifikaatori kasutamiseks mitteinfiksselt tuleb ta panna ümarsulgudesse. Näiteks `2 + (-3)` ja `(+) 2 (-3)` arvutatakse ühtmoodi. Õigupoolest esimese variandi kirjutab juba kompilaator ümber teise variandina. Tähelise identifikaatori kasutamiseks infiksselt tuleb ta panna tagurpidiülakomade vahele. Näiteks `mod 10 7` ja `10 `mod` 7` arvutatakse ühtmoodi — kompileerimise ajal asendatakse teine variant esimesega.

Funktsioonidele `div` ja `mod` on prelüüdis nende infiksse kasutamise jaoks ka prioriteet ja assotsiatiivsus defineeritud: mõlemad on vasakassotsiatiivsed ning korrutamise-jagamisega sama prioriteediga.

Kui on vaja arvutada samade arvude jaoks nii jagatis kui jääk, on efektiivsem kasutada funktsiooni `divMod`, mis võtab üksikargumentideks arvulised argumentid nagu ka `div` ja `mod`, kuid annab väärtuseks paari, milles on jagatis ja jääk. Analoogselt ka funktsioonide `take` ja `drop` väärtuse väljaarvutamisel samade argumentidega tehakse valdavalt sama tööd, mistõttu mõlema väljaarvutamise vajaduse korral on efektiivsem kasutada funktsiooni `splitAt`, mis võtab samad argumentid nagu `take` ja `drop`, kuid annab väärtuseks paari, mille komponentide väärtused on samad mis vastavalt funktsioonide `take` ja `drop` rakendamisel samadele argumentidele.

Ülesandeid

86. Leida list, mis koosneb järjekorras 43 vähimast kolmekohalisest 1-ga lõppevast arvust. Teha seda kahel viisil, millest üks saadakse teisest prefiksse ja infiksse kasutuse asendamisel üksteisega.
87. Kirjutada avaldis, mille väärtus on sama mis avaldisel `[1..2, 3..5]`, nii et ei kasuta ei listide erisüntakseid ega ühtki nime infiksselt.

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

Curried-kujul funktsioonide defineerimisel muutujate väärtuseks on muidugi võimalik kasutada siiani käsitletud viise. Näiteks kolme arvu aritmeetilist keskmist arvutava *curried*-kujul funktsiooni võib anda muutuja `aritm3` väärtuseks ükskõik kummaga deklaratsioonidest

```
aritm3 = \ a b c -> (a + b + c) / 3,  
aritm3 a  
  = \ b c -> (a + b + c) / 3.
```

Samas võib paremalt poolt ka rohkem argumentinäidiseid vasakule üle tuua, asetades nad nii nagu *curried*-kujul funktsiooni rakendamisel ikka:

```
aritm3 a b  
  = \ c -> (a + b + c) / 3'  
aritm3 a b c  
  = (a + b + c) / 3'
```

Niisugune argumentide liigutamine poolte vahel ei muuda deklaratsiooni tähendust.

Sama muutujat võib defineerida ka mitme deklaratsiooniga, mille vasakul pool on rohkem kui üks argumentinäidis. Sellisel juhul valitakse tegelike argumentide järgi esimene deklaratsioon, mille argumentinäidistega tegelikud argumentid kõik vastavalt sobituvad. Näiteks funktsiooni `risti`, mis võtab ükshaaval argumentideks kaks listi ja, kui need mõlemad on mittetühjad, annab välja esimese listi peast ja teise listi sabast koostatud listi, vastasel korral aga tühja listi, võib defineerida järgmiselt:

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

(17)

Esimene deklaratsioon läheb käiku parajasti juhul, kui mõlemad argumentid on mittetühjad listid, sest parajasti siis sobituvad mõlemad argumentid vastavate näidistega. Ülejäänud juhtudel läheb käiku teine deklaratsioon, sest seal argumentidele tingimusi ei seata.

Kui muutuja, mille väärtus on *curried*-kujul funktsioon, defineeritakse mitme deklaratsiooniga, lisandub nõue, et kõigis neis deklaratsioonides oleks vasakus pooles ühepalju argumentinäidiseid. Definiitsiooni (17) teise deklaratsiooni asendamine deklaratsiooniga

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

(18)

annab süntaksivea, kuigi üksikuna võttes on ta endise deklaratsiooniga samaväärne.

Muus osas kõik senivaadeldud reeglid kehtivad. Vasakus pooles võib defineeritavat muutujat kasutada isegi infiksselt.

Ülesandeid

89. Leida moodulist Prelude funktsiooni `const` definitsioon ja saada sellest aru.
90. Defineerida viimatiseletatud moel oma moodulis funktsioon `curriedDiff`, mis on ülesandes 64 defineeritud funktsiooni *curried*-kuju.
91. Defineerida eelmise ülesande funktsioon nii, et deklaratsiooni vasakus pooles oleks ta kasutatud infiksselt.
92. Kirjutada funktsioon `takeLõpust`, mis töötab nagu `take`, aga võtab elemente listi lõpust. Näiteks `takeLõpust 2 [3, 2, 4]` väärtustamine peab andma `[2, 4]`.
93. Kirjutada oma moodulisse muutuja `risti` definitsioon (17), kontrollida, et see töötab, ja asendada siis teine deklaratsioon deklaratsiooniga (18). Jätta meelde veeteade, mis tekib nüüd mooduli kompileerimisel: niisugune tekib, kui sama muutuja erinevad deklaratsioonid omavad vasakus pooles erinevat arvu argumentinäidiseid.
94. Kas definitsiooni (17) teise deklaratsiooni ärajätmisel saadav definitsioon on algsega samaväärne?
95. Kas definitsioon, mille saame definitsioonist (17) mõlema deklaratsiooni argumentinäidiste nihutamisel deklaratsiooni (18) eeskujul paremasse poolde, (a) kompileerub, (b) on algsega samaväärne?
96. Defineerida *curried*-kujul funktsioon, mis võtab ükshaaval argumentideks kaks listi ja annab väärtuseks listi, mille elementideks on kõik need objektid, mis esinevad esimese või teise listi esimese elemendina, igäüks täpselt üks kord.

Curried-kujul funktsioonid on funktsioonid, mille väärtused on omakorda funktsioonid. Haskellis saab muidugi ka funktsiooni argument olla funktsioon.

Funktsioone, mille argumentiks on funktsioon, nimetatakse **kõrgemat järku funktsioonideks** (ingl *higher-order function*). Niimoodi nimetatakse mõnikord ka funktsioone, mille väärtus on funktsioon, st *curried*-kujul funktsioone. Need kaks erinevat käsitlust saavad alguse erinevatest formaalsetest definitsioonidest, mis defineerivad väärtuse järgu naturaalarvuna. Järk loetakse kõrgemaks, kui ta on suurem kui 1. Mõlemal juhul võrdub väärtuse järk tema tüübi järguga. Mõlemal juhul on tüübi järk võrdne 0-ga parajasti siis, kui tegu pole funktsioonitüübiga. Öeldes teisiti, mittefunktsioonid on 0. järku funktsioonid, funktsioonid aga vähemalt 1. järku funktsioonid. Tähistades tüübi t järku ühes mõttes $\text{ord } t$ ja teises mõttes $\text{ord}' t$, saame, et nii $\text{ord } t = 0$ kui $\text{ord}' t = 0$ kehtivad parajasti siis, kui t ei esitu kujul $u \rightarrow v$. Kui aga $t = u \rightarrow v$ mingite tüüpide u ja v korral, siis $\text{ord } t = \max(\text{ord } u + 1, \text{ord } v)$, kuid $\text{ord}' t = \max(\text{ord}' u, \text{ord}' v) + 1$. Kui funktsiooni argument ja väärtus on mõlemad mittefunktsionaalsed, on mõlema definitsiooni järgi tegemist 1. järku funktsiooniga. Samas *curried*-kujul

funktsioonid on teise definitsiooni järgi juba kõrgemat järku funktsioonid (järk suurem kui 1), kui esimese definitsiooni järgi on näiteks + veel 1. järku funktsioon.

Kõrgemat järku funktsioonid on funktsionaalses programmeerimises väga olulised, andes ühe võimaluse koodi abstraktsioonitaseme tõstmiseks ja koodikorduse vältimiseks. Seda näeme lähemalt selles kursuses hiljem; praegu teeme ainult põgusa n-ö ametliku tutvuse, kuna funktsioone argumendiks võtvate funktsioonide kasutamine ega defineerimine ei seonu ühegi uue süntaktilise konstruktsiooni ega reeglga.

Eelnevalt funktsioonide prefiksse kasutusega seoses tuli välja prefiksset rakendamist märkiva järjestkirjutamise võimalik käsitus vasakassotsiatiivse infiksoperaatorina prioriteediga 10. Kui nii, siis prefiksne rakendamine on kõrgemat järku funktsioon, kuna ta võtab argumendiks funktsiooni.

Prefiksse rakendamise jaoks on prelüüdis defineeritud ka üks päris infiksoperaator \$. Nii näiteks avaldise `log 5` kohal võib samaväärselt kirjutada `log $ 5`, avaldise `take 2 [5, 6, 7]` kohal aga `take 2 $ [5, 6, 7]` või isegi `(take $ 2) $ [5, 6, 7]`.

Arvestada tuleb aga prioriteedi ja assotsiatiivsusega: operaatoril \$ on need tavalise järjestkirjutamisega võrreldes vastupidised. Ta on paremassotsiatiivne prioriteediga 0. See on kasulik, kuna erinevates situatsioonides võib mugavam olla nii üks kui teine äärmus. Näiteks võtmaks avaldisega `[-100, -97 .. 0] ++ [1, 4 .. 100]` määratud listist vahejuppi 21. kuni 40. elemendini, võib kirjutada avaldise

```
take 20 $ drop 20 $ [-100, -97 .. 0] ++ [1, 4 .. 100].
```

Ilma \$-operatsioonita tulnuks kasutada sulgusid, mis tekitanuks mõnevõrra halvemini loetava avaldise

```
take 20 (drop 20 ([-100, -97 .. 0] ++ [1, 4 .. 100])).
```

Operaatori \$ tähenduses pole mingit sisulist erinevust tavalisest funktsioonirakendamisest. Kirjutis kujul `f $ x` kirjutatakse lihtsalt ümber kujul `f x`. Haskellis on olemas veel üks funktsioonirakendamisoperaator \$!, mis erinevalt senistest väärtustab kõigepealt argumendi kuni välimise konstruktori ilmumiseni või vea tekkimiseni (funktsiooni puhul senikaua, kui ilma argumendita on võimalik väärtustada) ja alles seejärel rakendab tulemusele funktsiooni. Prioriteet ja assotsiatiivsus on tal samad mis operaatoril \$.

Operaatori \$! abil on võimalik vajadusel funktsioone agaraks sundida. Näiteks kui avaldised `const 5 (error "E")` ja `const 5 $ error "E"` annavad mõlemad väärtustamisel tulemuseks 5, siis `const 5 $! error "E"` tekitab väärtustamisel täitmisaegse vea veateatega "E". Kui `error "E"` asemele panna `length [1 ..]`, siis esimesed kaks varianti annavad ikka väärtuseks 5, viimane variant aga jääb lõpmatusse tsüklisse.

Operaatori \$! erinevust operaatorist \$ näitab väga ilmekalt avaldiste

```
error "fun" $! error "arg" (19)
```

ja

$$\text{error "fun" } \$ \text{ error "arg"} \quad (20)$$

väärtustamine. Kui (19) väärtustamisel tuleb välja argumendi viga, siis (20) väärtustamisel funktsiooni viga. Sama lugu on mistahes vigaste või lõpmatusse arvutusse minevate funktsiooni ja argumendi korral.

Olgu lõpuks toodud ka üks näide kõrgemat järku funktsiooni defineerimisest muutuja väärtuseks. Järgnev deklaratsioon seob muutujaga `kõrgem` funktsiooni, mis võtab argumendiks funktsiooni ja annab väärtuseks selle funktsiooni väärtuse kohal 0:

$$\begin{aligned} \text{kõrgem } f \\ = f \ 0 \end{aligned} \quad (21)$$

Nüüd näiteks `kõrgem cos` annab väärtuseks arvu 1, sest `kõrgem cos = cos 0 = 1`. Samuti on `kõrgem (2 +)` ja `kõrgem (subtract 5)` korrektsed avaldised: esimese väärtus on 2, teise väärtus -5 .

Funktsiooni `kõrgem` võib rakendada ka *curried*-kujul funktsioonidele, sest ainus tingimus, mida definitsioon (21) funktsiooni `kõrgem` argumendile tegelikult seab, on see, et see argument võtaks argumendiks arvulise objekti. Näiteks aritmeetikatehted kõik sobivad. Nii näiteks on korrektne avaldis `kõrgem (*)`, mille väärtuseks on nulliga korrutatav funktsioon ehk konstantne 0. Iga infiksoperaatori \oplus korral on avaldis `kõrgem \oplus` väärtuselt võrdne sektsiooniga `(0 \oplus)`.

Ülesandeid

97. Mitmendat järku funktsioon on kummagi käsitluse kohaselt (a) eelnevalt defineeritud funktsioon `aritm3`, (b) `$`, kui ta on rakendatud argumentidele `log` ja `5`?
98. Leida moodulist Prelude operaatori `$` definitsioon ja saada sellest aru.
99. Defineerida oma moodulis muutuja `?` väärtuseks postfiksne funktsioonirakendamine (funktsioon oma argumendi järel, st argument esimese ja funktsioon teise argumendina).
100. Teha läbi avaldiste `kõrgem (2 +)` ja `kõrgem (subtract 5)` sammsammuline väärtustamisprotsess ja veenduda, et tulemuseks on vastavalt 2 ja -5 . (`kõrgem` on defineeritud deklaratsiooniga (21).)
101. Kas avaldis `kõrgem (: [88])` on tüübikorrektne? Kui jah, siis leida selle avaldise väärtus, tehes läbi sammsammulise väärtustamisprotsessi. (`kõrgem` on defineeritud deklaratsiooniga (21).)
102. Kas avaldis `kõrgem take` on tüübikorrektne? Kui jah, siis kirjeldada selle avaldise väärtust. (`kõrgem` on defineeritud deklaratsiooniga (21).)

Rekursioon

Rekursiivseks (ingl *recursive*) nimetatakse definitsiooni, mille parem pool sisaldab muutujat, mida see definitsioon defineerib. 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.

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

Kuna muutujate väärtuse muutmise on võimatu, ei saa Haskellis olemas olla ka imperatiivsetes keeltes tavapäraseid tsikleid. Kõik tsüklilised protsessid tulebki programmeerida rekursiooniga.

Selle osa jaoks võiks teha uue faili, nt nimega `Rec.hs`.

Rekursiivselt defineeritud funktsioonid

Funktsiooni rekursiivsel defineerimisel tuleb funktsiooni käitumine suuremate argumentide korral avaldada funktsiooni väärtuste kaudu väiksematel argumentidel. Tihti on vaja spetsifitseerida ka baasjuht või -juhud, milles rekursiivset pöördumist ei toimu. Kui baasjuhud puuduvad, toimuvad rekursiivsed pöördumised lõputult — lõpmatu listi arvutamisel see ongi eesmärk.

Seejuures suurem-väiksem seos on erinevatel juhtudel erinev. See võib kokku langeda arvude suurusjärjestusega — see tähendab, et funktsiooni argumendid on arvulised. Baasjuhiks on siis tüüpiliselt funktsiooni käitumine argumendil 0. Samas võib tegu olla struktuurse rekursiooniga, kus funktsiooni käitumine suurematel andmestruktuuridel spetsifitseeritakse funktsiooni väärtuste kaudu väiksematel andmestruktuuridel. Andmestruktuuriks on väga tihti just listid. Sellisel juhul definitsioon tüüpiliselt sätestab funktsiooni väärtuse mittetühjal listil funktsiooni väärtuse kaudu selle listi sabal ning tüüpilise baasjuhuna säteatakse funktsiooni väärtus tühjal listil.

Rekursiivselt defineeritava täisarvuliste argumentidega funktsiooni tüüpilisteks näideteks on mõne rekurrentselt defineeritud jada liikmeid arvutavad funktsioonid. Üks tuntumaid rekurrentselt defineeritud funktsioone on faktoriaal. Matemaatilisel antakse ta seostega

$$0! = 1, \quad \forall n > 0 (n! = (n - 1)! \cdot n).$$

Selle definitsiooni saab sisuliselt otse ümber kirjutada Haskellis koodiks. Loenguslaididel on antud rida faktoriaali definitsioone, millest enamik on otseselt rekursiivsed ning need, mis ei ole, kasutavad kaudselt rekursiooni, st kasutavad standardteegi muutujaid, mille definitsioon on rekursiivne või kasutab omakorda kaudselt rekursiooni.

Kui funktsioonile antud matemaatiline spetsifikatsioon ei ole rekurrentne, kuid tema arvutamine nõuab tsüklilist protsessi, peab rekurrentse seose ise tuletama. Olgu meil näiteks vaja programmeerida funktsioon f matemaatilise definitsiooniga

$$f(n) = \sum_{i=1}^n i^4 = 1^4 + \dots + n^4,$$

st $f(n)$ on n esimese positiivse täisarvu 4. astmete summa. Tuleb leida rekurrentne seos $f(n)$ ja $f(n - 1)$ vahel, milleks on

$$f(n) = f(n - 1) + n^4, \tag{22}$$

sest selleks, et $f(n - 1)$ -st ehk esimese $n - 1$ positiivse täisarvu 4. astmete summast saada $f(n)$ ehk esimese n positiivse täisarvu 4. astmete summa, piisab talle liita puuduv liige n^4 . On vaja ka rekursiooni baasjuhtu — selleks võtame vähima võimaliku argumendi, mille korral antud funktsiooni spetsifikatsioon veel mõtet omab, ehk $n = 0$; ilmselt sobib $f(0) = 0$ (siis kehtib seos (22) ka $n = 0$ korral). Lisades veateate andmise negatiivse argumendi puhul, saame Haskell'i definitsiooniks

```

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

```

. (23)

Iga funktsioon, mis antakse argumendil n kui summa mingi teise funktsiooni väärtustest argumentidel 1-st n -ni, on analoogiliselt 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. Sama kehtib ka korrutamise korral, ainult et argumendil 0 on funktsiooni väärtus 1 — korrutamise ühikelement. Faktoriaal näiteks on sellest erijuht, kus argumendil n korrutatakse kokku identsusfunktsiooni väärtused argumentidel 1-st n -ni.

Rekurrentse võrrandi matemaatilise spetsifikatsiooni otsene tõlge Haskell'i koodiks on kasutatav ainult 1. järku võrrandite puhul. Kõrgemat järku rekurrentse võrrandi puhul annab see naiivne lähenemine eksponentsiaalse keerukusega töötava koodi, seetõttu tuleb kasutada kavalamaid meetodeid.

Võtame näiteks Fibonacci jada, mis matemaatiliselt antakse seostega

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

Võrrand on 2. järku, sest elemendid on määratud 2 eelneva kaudu. See jada üldistub loomuldasa ka negatiivsetele indeksitele valemiga

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

Kui kirjutaksime Haskell'i koodi otse matemaatiliste seoste põhjal, võiksime saada näiteks defintsiooni

```
fib n
  | n >= 2
  = fib (n - 1) + fib (n - 2)
  | n >= 0
  = n
  | otherwise
  = (if odd n then 1 else -1) * fib (-n)
```

 (24)

Kasutatud standardfunktsioon `odd` kontrollib, kas argument on paaritu arv. Selle koodi järgi arvutades toob iga `fib` väljakutse 1-st suuremal argumendil kaasa kaks uut väljakutset, väljakutsete arv kasvab eksponentsiaalselt. Lineaarse keerukuse saavutamiseks tuleks pidevalt hoida kaks viimast vahetulemust korraga kättesaadavana, igal tasemel peab piisama ühekordsest rekursiivsest väljakutsest.

Üks võimalus selleks on defineerida abifunktsioon, mille väärtuseks oleksid paarid kahest järjestikusest Fibonacci arvust, ja anda otsitava funktsiooni väärtus kui abifunktsiooni töö tulemuse üks komponent:

```
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)
```

 (25)

Ülesandeid

103. Kirjutada oma moodulis veel üks faktoriaalifunktsiooni `fact` defintsioon, mis kasutab

funktsiooni `compare`; negatiivse argumenti puhul peab ta töö lõpetama olukorda identifitseeriva veateatega.

104. Kontrollida interaktiivse interpretaatoriga, et koodiga (25) defineeritud `fib` töötab vastuvõetava ajaga märksa suuremate argumentide korral kui koodiga (24) defineeritud `fib`.

Kõik funktsioonid listidel, mis teevad midagi määramata arvu komponentidega, tuleb programmeerida rekursiooniga. Loenguslaididel on rida näiteid ka sellest.

Eeldefineeritud funktsioonide abil on Haskellis võimalik väga palju kasulikke arvutusi ära realiseerida. Siiski ei ole vaid eeldefineeritud funktsioone kasutatav arvutus alati kõige efektiivsem. Võtame näiteks avaldised kujul

```
(elem a l , delete a l). (26)
```

Juba tuttav funktsioon `elem` võtab objekti ja sama tüüpi elementidega listi ja annab väärtuseks `True` parajasti siis, kui antud objekt esineb antud listis. Moodulis `Data:List` defineeritakse funktsioon `delete`, mis võtab samasugused argumentid ja annab väärtuseks listi, mille saab argumentidest argumentobjekti esimese esinemise eemaldamisel, kui argumentobjekt argumentlistis esineb, vastasel korral annab väärtuseks listi enda. Väärtustades avaldist (26), otsivad mõlemad funktsioonid sama objekti esinemist samast listist ja seda tööd teeb kumbki teisest sõltumatult ise. Seega kui mõne objekti ja listi jaoks on vaja rakendada nii funktsiooni `elem` kui funktsiooni `delete`, mis võib praktikas kergesti ette tulla, on mõtet kirjutada uus funktsioon, mis arvutab soovitud tõeväärtuse ja listi ühekorraga ning annab nad paarina vastuseks. Kood võiks olla

```
eemaldaEsimene a (x : xs)
  | a == x
  = (True , xs)
  | otherwise
  = let
      (tv , us) = eemaldaEsimene a xs
    in
      (tv , x : us)
eemaldaEsimene _ _
  = (False , []) (27)
```

Defineerime siin paaride kasutamise näitena veel funktsiooni `split2`, mis võtab argumentiks listi ja jaotab tema elemendid üle ühe kahte listi, andes tulemuseks listipaari:

```
split2 (x : xs)
  = let
      (us , vs) = split2 xs
    in
      (x : vs , us)
split2 _
  = ([] , []) (28)
```

Ülesandeid

105. Leida moodulist Prelude funktsioonide `!!`, `take`, `drop`, `splitAt` definitsioonid ja saada neist aru.
106. Kirjutada oma moodulisse definitsioon (28) ja veenduda interaktiivse interpretaatori käsurida kasutades, et defineeritud funktsioon `split2` lõpmatul argumentlistil on paar kahest lõpmatust listist.
107. Kirjutada *curried*-kujul funktsioon `eemaldaKõik`, mis tagastab argumentobjektile x ja argumentlistil l paari, mille esimene komponent on tõeväärtus, mis ütleb, kas x esineb listis l , ja teine komponent saadakse l -st kõigi x esinemiste eemaldamisel.
108. Defineerida *curried*-kujul funktsioon `flatZip`, mis võtab argumendiks kaks listi ja koostab 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. Sisuliselt on tegu koodiga (28) defineeritud funktsiooni `split2` järelpöördoperatsiooni *curried*-kujuga.

Mõnikord ei piisa listirekursiooni puhul sellest, et programmeeritakse rekursiooni baas vaid tühja listi jaoks. On vaja eraldi spetsifitseerida ka funktsiooni käitumine üheelemendilise listi jaoks. Niisugune on näiteks funktsioon, mis kontrollib, kas kõik elemendid listis on võrdsed. Anname sellele kaks võimalikku näitedefinitsiooni.

Esimene on otsene definitsioon:

```
kõikVõrdsed (x : xs@ (y : _))
  = x == y && kõikVõrdsed xs
kõikVõrdsed _
  = True
```

(29)

Kuna funktsiooni väärtus tühjal ja üheelemendilisel listil on sama, õnnestub need kaks juhtu kokku võtta ja piirduda kahe deklaratsiooniga. Algoritmi üldpõhimõtteks on kontrollida, kas iga kaks järjestikust elementi on võrdsed.

Teine definitsioon

```
kõikVõrdsed (x : xs)
  = let
    kõikVõrdsed (z : zs)
      = x == z && kõikVõrdsed zs
    kõikVõrdsed _
      = True
  in
    kõikVõrdsed xs
kõikVõrdsed _
  = True
```

(30)

saab hakkama mittetühja listi juhu ühtse kirjeldamisega, kuid kasutab selleks abifunktsiooni, mille kutsub välja argumentlisti sabal ja mille defineerib omakorda rekursiooniga. Algoritmi üldpõhimõtteks on kontrollida, kas kõik elemendid võrduvad esimesega.

Definitsioon (30) näitab muuhulgas seda, et lokaalselt saab varjata isegi sellesamas deklaratsioonis defineeritava muutuja. Definitsiooni (30) esimene deklaratsioon defineerib globaalset muutajat kõikVõrdsed, kuid defineerib lokaalselt samanimelise muutuja kõikVõrdsed, mille väärtus on hoopis teine. Lokaalne kõikVõrdsed on nähtav *let*-avaldise piires.

Järgnevalt defineeritud funktsioon `eraldaSegment` võtab argumentiks listi ja lõhub selle kaheks listiks esimeselt kohalt, kus elemendid ei lähe mittekahanevalt:

```

eraldaSegment (x : xs)
  = let
    eraldaSegment x xs@ ~(y : ys)
      | null xs || x > y
      = ([x] , xs)
      | otherwise
      = let
          (us , vs) = eraldaSegment y ys
        in
          (x : us , vs)
    in
    eraldaSegment x xs
eraldaSegment _
  = ([] , [])

```

(31)

Selle abil saab defineerida funktsiooni `segmendid`, mis antud listi järgi teeb tema segmentide listi, st tulemuslisti elementideks on maksimaalsed mittekahanevad lõigud argumentlistis:

```

segmendid []
  = []
segmendid xs
  = let
    (us , vs) = eraldaSegment xs
  in
    us : segmendid vs

```

(32)

Definitsioon (32) on küll rekursiivne, sest teine deklaratsioon sisaldab pöördumist enda poole, kuid rekursiooniskeem on ebatavaline, kuna rekursiivne pöördumine ei toimu listi saba poole. Kui argument on mittetühi list — käiku läheb siis teine deklaratsioon, sest esimese deklaratsiooni argumentinäidisega ta ei sobitu —, kutsutakse välja funktsioon `eraldaSegment`, mis annab kätte esimese segmendi ning järelejääva osa — rekursiivne pöördumine tehakse viimasele.

Ülesandeid

109. Näidata, millised nime kõikvõrdsed esinemised definitsioonis (30) tähistavad globaalset ja millised lokaalset muutujat.
110. Kirjutada funktsioon järjestatud, mis võtab argumendiks arvude listi ja annab väärtuseks True või False vastavalt sellele, kas list on lõplik ja mittekahanevalt järjestatud või leidub listis element, mis on talle eelnevast väiksem.
111. Kirjutada funktsioon vahed, mis leiab etteantud arvude listi järgi listi, mis koosneb argumentlisti kahe järjestikuse elemendi vahedest.
112. Kirjutada funktsioon kuniKorduseni, 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.
113. Kas definitsioonis (32) esimese deklaratsiooni ärajätmisel saadud kood defineerib sama funktsiooni kui algne kood?
114. Kirjutada ülesandes 107 defineeritud funktsiooni eemaldaKõik abil funktsioon korduvad, mis võtab argumendiks listi ja annab tulemuseks paarikaupa erinevate elementidega listi, mis koosneb parajasti argumentlistis korduvatest elementidest.

Vastastikku rekursiivsete funktsioonide korral võib töö käigus tsükliline tagasipöördumine sama funktsiooni poole aset leida mõne teise funktsiooni väljakutse kaudu. Näiteks deklaratsioonidega

```
üksnul 0
    = ""
üksnul n
    = '1' : nulüks (n - 1)

nulüks 0
    = ""
nulüks n
    = '0' : üksnul (n - 1)
```

defineeritud funktsioonid üksnul ja nulüks on omavahel vastastikku rekursiivsed, sest nad kutsuvad kumbki teineteist välja.

Ülesandeid

115. Mis on funktsiooni üksnul väärtus mittenegatiivsel argumendil n ? Mõistatage peast ja siis testige arvutil.

Kui meile pakub näiteks huvi ainult üksnul ja funktsiooni nulüks ainus väljakutse asub üksnul kehas, võib nulüks definitsiooni viia *let*-avaldisega üksnul lokaalseks definitsioo-

niks. Niisiis võivad Haskellis vastastikku rekursiivsed olla ka funktsioon ja tema kehas lokaalselt defineeritud funktsioon.

Ülesandeid

116. Kirjutada funktsioon `vaheLdub`, 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.

Kui rekursiivne protsess kutsub välja mingi arvutuse, mis ei sõltu rekursiooni parameetritest, tuleks see arvutus viia rekursioonist välja, et seda sooritataks nii vähe kordi kui võimalik.

Akumulaatorid

Akumulaator (ingl *accumulator*) on rekursiivse funktsiooni formaalne argument, milles väärtustamise käigus asuvad arvutuste vahetulemused.

See tähendab, et rekursiivsel pöördumisel antakse uuendatud vahetulemus selles argumendis kaasa. Tihti on akumulaatoriteks spetsiaalselt selle eesmärgiga loodud lisaargumendid.

Akumulaatoreid võib kasutada üsna mitmel eesmärgil. Üks neist võib olla lihtsalt imperatiivse programmeerimisstiili järgmine. Imperatiivses programmeerimises kujuneb arvutuse lõpptulemus tüüpiliselt välja mingi perioodiliselt uuendatud väärtusega muutujas. Kuna funktsionaalses programmeerimises muutujate väärtused plokki sees ei muutu, saab sellist arvutustöökäiku lavastada vaid rekursiivsete funktsioonide formaalsete argumentidega, mis saavad võtta uue väärtuse igal järjekordsel pöördumisel funktsiooni poole.

Funktsionaalses keeles näeb imperatiivset paradigmat ahviv programm muidugi välja üsna teistmoodi kui imperatiivses keeles, taoline stiil koodi loetavust vaevalt et parandab. Akumulaatori kasutamise sisulistest eesmärkidest kõige levinum on pikkade väärtustamata avaldiste vältimine arvutuse käigus ja seega mälu kokkuhoid.

Anname koodiga (23) defineeritud funktsioonile suurSumma uue definitsiooni

```
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"      (33)
```

Juhud, kus argument on mittenegatiivne, on kokku võetud ja realiseeritud endisest erinevalt, akumulaatoriga lokaalse abifunktsiooniga, mille nimi on samuti suurSumma (lihtsuse mõttes, et ei peaks uut nime välja mõtlema).

Lokaalse abifunktsiooni definitsiooni saab kõigepealt võtta kui imperatiivse programmeerimisstiili järgimist, sest arvutus toimub sarnaselt sellele, mida imperatiivses keeles 4. astmete summat arvutava funktsiooni programmeerimisel tehtaks tsükli sees. Lokaalse suurSumma esimene argument *a* on akumulaator, tema väärtustena jooksevad arvutusprotsessi käigus läbi vahetulemused, teine argument *i* aga mängib tsükliindeksi rolli. Valvur $i \leq n$ täidab tsükli jätkamistingimuse rolli, sest parajasti siis, kui tema annab `True`, toimub rekursiivne pöördumine. Algselt on akumulaatori väärtus 0, sest väline suurSumma kutsub *let*-avaldises välja sisemise suurSumma argumendiga 0. Igal rekursiivsel pöördumisel antakse suurSumma uueks akumulaatorargumendiks akumulaatori jooksva väärtuse ja jooksva tsükliindeksi 4. astme summa ning tsükliindeksiks jooksvast tsükliindeksist järgmine naturaalarv. Niimoodi tekib akumulaatorisse 1^4 liitmine, 2^4 liitmine jne. Kui *i* väärtus on saanud suuremaks *n* väärtusest ehk kõik väärtused 1-st kuni *n* väärtuseni on läbitud, antakse funktsiooni väärtusena välja akumulaator, kuhu ongi töö käigus kogunenud just vajalikud liitmised.

Samas definitsioonis on lihtsasti märgatav ka akumulaatori kasutamise teine eesmärk. Varasema definitsiooni (23) järgi väärtustades ei teki enne rekursiooni põhjani jõudmist ühtki summaavaldist, mis ei sisaldaks muutujat suurSumma, mistõttu liitmiste sooritamine saab põhimõtteliselt alata alles rekursioonisügavusest tagasipöördumisel, enne tuleb aga kõik need liitmised pika avaldisena mällu mahutada. Uues definitsioonis on aga olemas summaavaldis, kus liidetavad tundmatuid ei sisalda — nimelt akumulaator rekursiivsel väljakutsel — ja mida on seega võimalik jooksvalt väärtustada, et mälu säästa.

Katsetades selgub aga, et definitsioon (33) praktiliselt polegi parem kui definitsioon (23). Suurim argumendi väärtus, mille korral suurSumma arvutamine mälu ületäitumist ei põhjusta, on definitsiooni (33) kasutades enam-vähem sama mis definitsiooni (23) puhul. Asi on selles, et Haskell väärtustab vaikimisi kõike laisalt ja kuna definitsioonis (33) pole midagi, mis nõuaks

akumulaatori väärtustamist, siis seda ka ei toimu ja akumulaator kogub endasse vaid järjest pikenevat väärtustamata summaavaldist. Et olukorda parandada, tuleb akumulaatorit operaatoriga `!` jooksvalt väärtustama sundida. Piisab see operaator sobivasse kohta lisada:

```

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"

```

(34)

Avaldis `(!) suurSumma (a + i ^ 4)`, mis on muidugi samaväärne avaldisega `suurSumma ! a + i ^ 4`, paneb arvutusprotsessi iga kord enne `suurSumma` rekursiivset väljakutset avaldist `a + i ^ 4` väärtustama. Niimoodi defineeritud funktsiooni võib edukalt kasutada mitu suurusjärku suuremate argumentide korral kui eelmisi versioone.

Seejuures avaldist `i + 1` see `!` väärtustama ei sunni, kuid seda polegi vaja, kuna see avaldis väärtustatakse kohe pärast rekursiivset pöördumist esimest valvurit väärtustades niikuinii.

Ülesandeid

117. Kirjutada oma moodulisse deklaratsioonid (23), (33), (34) ja neid kahekaupa välja kommenteerides kontrollida igäihe puhul, kui suure argumentiga on Hugs võimeline niimoodi defineeritud funktsiooni arvutama ilma mälu ületäitumiseta.

118. Arvu x kahanevaks faktoriaaliks naturaalarvu k järgi nimetatakse arvu

$$(x)_k = x(x - 1) \dots (x - (k - 1)).$$

Anda oma moodulis akumulaatoriga definitsioon *curried*-kujul funktsioonile `kfact`, mille väärtus argumentidel x ja k on $(x)_k$. Akumulaatoris peab toimuma jooksev väärtustamine. Negatiivse teise argumenti korral peab ta andma vastavasisulise veateate.

Akumulaatorid on mõnikord kasulikud ka ajalise keerukuse vähendamiseks. Näiteks vaatleme loengumaterjalis toodud näidet listi ümberpööramisest. Naiivne definitsioon

```

reverse (x : xs)
  = reverse xs ++ [x]
reverse _
  = []

```

(35)

viib ruutkeerukusega arvutuseni, sest kui tähistada n -ga argumentlisti pikkust, siis protsessi käigus vasakult konkateneeritavate listide pikkused on $0, 1, \dots, n - 1$, mille summa on ruutpolünoom n suhtes. Akumulaatori kasutamine võimaldab liste pöörata lineaarse keerukusega:

```
reverse
= let
  reverse as (x : xs)
    = reverse (x : as) xs
  reverse as _
    = as
in
reverse []
```

Siin on akumulaatoriks lokaalse funktsiooni argument `as`.

Ka kõrgemat järku rekurrentsete võrranditega antud jadade liikmete efektiivseks arvutamiseks võib kasutada akumulaatoreid. Eelmises jaotises vaadeldud viis Fibonacci jada liikmete leidmiseks lineaarse keerukusega pole ka parim, sest igal rekursioonisammul konstrueeritakse uus paar, andmestruktuuri tekitamine mälus aga nõuab lisaressurssi. Selle asemel võib korraldada summade leidmist akumulaatoris. Et kogu aeg on vaja teada 2 vahetulemust, peab ka akumulaatoreid olema 2. Tulemus on järgmine:

```
fib n
| n >= 0
  = let
    fib a _ 0
      = a
    fib a b n
      = fib b (a + b) (n - 1)
  in
    fib 0 1 n
| otherwise
  = (if odd n then 1 else -1) * fib (-n)
```

(36)

Koodi (36) eelis arvutuskiiruses koodiga (25) võrreldes on siiski vaid marginaalne.

Ülesandeid

119. Mängida paberil läbi avaldise `fib 5` väärtustusprotsess definitsiooniga 36.

120. Anda koodiga (28) defineeritud funktsioonile `split2` kahe akumulaatoriga definitsioon.

121. Kirjutada definitsiooni (36) põhjal `fib` selline definitsioon, mille järgi arvutades toimub akumulaatoris jooksev väärtustamine.

On veel üks tüüpiline olukord, kus akumulaatoritehnika on hädavajalik — lõpmatute listide elementide omavaheliste suhete analüüsimine.

Olgu meil näiteks vaja mistahes listi kohta kontrollida, ega temas ei ole korduvaid elemente. Pidades silmas vaid lõplikke liste, võib programmeerija kirjutada definitsiooni

```
kordusteta (x : xs)
  = not (elem x xs) && kordusteta xs
kordusteta _
  = True
```

 (37)

Kui list on tühi, antakse `True`, sest tühjas listis ükski element ei kordu. Mittetühja listi puhul peab elementide mittekorduvuseks olema täidetud parajasti kaks tingimust: et esimene element ei esine järgmiste hulgas (kontrollib `not (elem x xs)`) ja et nende järgmiste elementide hulgas ei esine kordumisi (kontrollib `kordusteta xs` rekursiivselt). Niisiis tundub, et definitsioon on korrektne.

Paraku see definitsioon ei kõlba kuskile, kui argumentlistid ei pruugi olla lõplikud. Näiteks lõpmatul listil, mille esimene element on 0 ja kõik ülejäänud elemendid võrduvad 1-ga, jääb funktsioon lõpmatult tööle, kuna algoritm, mille definitsioon (37) realiseerib, kõigepealt kontrollib, kas esimene element tagapool esineb — kuna tagapool on lõpmata palju elemente ja neist ükski esimesega ei võrdu, jääb see osa lõpmatult tööle. Samas on selge, et piisaks vaid teise ja kolmanda elemendi võrdlusest tegemaks kordumiste esinemine kindlaks.

Algoritm, mis siinkohal aitaks, peaks tegema võrdlusi teises järjekorras. Parem algoritm võrdleb iga elementi temast eespool olevatega, mitte tagapool olevatega nagu teeb (37). Igale elemendile eelneb listis vaid lõplik arv elemente ja seetõttu jõuab niisugune algoritm suvalise kahe elemendi võrdlemiseni lõpliku arvu võrdluste järel.

Definitsioon

```
kordusteta
  = let
    kordusteta as (x : xs)
      = not (elem x as) && kordusteta (x : as) xs
    kordusteta _ _
      = True
  in
  kordusteta [ ]
```

 (38)

realiseeribki korduvuste leidmise sellise algoritmiga. Akumulaatoris `as` hoitakse jooksvalt argumentlisti alguselemente, mille omavahelised võrdlused on juba tehtud. See definitsioon leiab esineva korduse üles isegi juhul, kui list on osaline.

Ülesandeid

122. Kui definitsioonis (38) konjunktsiooni pooled ära vahetada, kas definitsioon muutuks halvemaks, paremaks või pole vahet?
123. Kirjutada funktsioon `disjunktssed`, mis võtab ükshaaval 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?
124. Kirjutada funktsioon `uuriKolmikud`, mis võtab argumendiks listi ja annab välja `True`, kui listis leidub kolm erinevas positsioonis elementi, mis moodustavad aritmeetilise progressiooni, vastasel korral kui list on lõplik, siis annab välja `False`.

“Jaga ja valitse” tehnika

“Jaga ja valitse” (ingl *divide and conquer*) tehnika puhul jagatakse argument igal rekursioonisammul kaheks võimalikult võrdseks “osaks” ja kombineeritakse otsitav väärtus rekursiivse pöördumise tulemustest neil osadel. Mõnikord võimaldab see keerukuses võita.

Olgu meil näiteks vaja programmeerida tavaline arvu astendamise täisarvuga, mis positiivsel astendajal defineeritaks korduva korrutamise kaudu, negatiivsel aga lisaks pöördarvu leidmise abil. Kandes matemaatilise definitsiooni naiivselt üle Haskellis, saaksime definitsiooni

```
aste a n
= case compare n 0 of
  GT
    -> aste a (n - 1) * a
  EQ
    -> 1
  _
    -> 1 / aste a (- n)
```

(39)

Arvutus selle definitsiooni järgi sooritab positiivse astendaja korral niipalju operatsioone, kui suur on astendaja väärtus. Teisi sõnu, see definitsioon töötab lineaarse keerukusega astendaja suhtes. Samas saab korduva ruututõstmisega korrutamiste arvu oluliselt vähendada, sest nii leiame sama arvu astmed logaritmilise korrutamiste arvuga. Näiteks a^8 leidmisel pole mõtet rakendada arvutusskeemi $1 \xrightarrow{a} a \xrightarrow{a} a^2 \xrightarrow{a} a^3 \xrightarrow{a} a^4 \xrightarrow{a} a^5 \xrightarrow{a} a^6 \xrightarrow{a} a^7 \xrightarrow{a} a^8$, sooritades 8 korrutamist, kuna lihtsam on leida sama väärtus skeemi $a \xrightarrow{\text{sqr}} a^2 \xrightarrow{\text{sqr}} a^4 \xrightarrow{\text{sqr}} a^8$ järgi, kus `sqr` tähistab ruututõstmist, tehes vaid 3 korrutamist.

See kaval viis astme arvutamiseks rakendab “jaga ja valitse” tehnikat. Arvu astme arvutamiseks leitakse samal meetodil sama arv poole väiksemas astmes ning tõstetakse tulemus ruutu. Kujutades astet ette pika korrutisena, jagab see viis astme arvutamiseks tegurid kahte võrdsesse gruppi,

leiab samal viisil tulemuse kummalgi gruppidest ning korrutab need. Võit keerukuses tuleb sel-
lest, et kuna grupid on võrdsed, piisab algoritmi rakendada vaid ühel neist, tulemuseks teisel
grupil võib arvutusi tegemata võtta sama arvu.

Üldjuhul tuleb veel käituda pisut erinevalt paaris ja paaritute astendajate korral, sest paaritut
tegurite arvu ei õnnestu täpselt pooleks jagada, umbes pooleks jagamine aga antud juhul võitu
ei anna. Paaritute astendajate puhul tuleb üks tegur eraldada, nii et järele jääb paarisarv tegureid,
viimastega käituda juba kirjeldatud moel ning tulemust lõpuks korrutada algul eraldatud teguriga.
Saame definitsiooni

```

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)

```

(40)

mis erineb definitsioonist (39) vaid positiivse astendaja juhul.

Võtmesammuks definitsioonis (40) on rekursiivse pöördumise $aste\ a\ q$ sidumine lokaalse
muutujaga, mille tulemusel toimub igal rekursioonitasemel vaid üks rekursiivne pöördumine ja
vaheaste arvutatakse tõepoolest vaid ühe korra välja. Kui seda mitte teha ja **in** järel olevas aval-
dises panna z asemele $aste\ a\ q$, oleks kogu ümbertöötamise vaev kasutu, korrutamiste arv
võrduks vanaviisi astendaja väärtusega.

Ülesandeid

125. Lülitada Hugsis sisse reduktsioonide (arvutussammude) ja kasutatud mäluühikute loendur
(:s +s) ning arvutada ratsionaalarvude suuri astmeid nii definitsiooniga (39), definitsioo-
niga (40) 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 ressursi kui definitsioon (39), samas kui (40) järgi arvutamine on
tunduvalt kiirem ja piirdub üliväikese reduktsioonide arvuga.

Klassikalised näited “jaga ja valitse” tehnika rakendamisest on listi sorteerimine kiir- ja mesti-
mismeetodil.

Kiirmeetodi põhimõtteks on jagada listi elemendid kaheks nii, et esimeses grupis on kõik ele-
mendid väiksemad teise grupi kõigist elementidest. Siis sorteeritakse samal meetodil kumbki

grupp ning tulemus saadakse esimese grupi elementide sorteeritud listi konkateneerimisel teise grupi elementide sorteeritud listi ette. Kui jagamisetapil tekkivad grupid on üldjuhul üksteisele lähedase suurusega, on kiirmeetod keerukusklassis $n \log n$ listi pikkuse n suhtes. Kui aga domineerivad ebaühtlased jagunemised, siis kiirmeetod töötab ruutkeerukusega ja parem on kasutada mõnd muud meetodit.

Haskellis selle algoritmi realiseerimiseks kirjutame kõigepealt funktsiooni, mis teostab listi jagamist kaheks nõutud viisil. Seda sobib tegema definitsioon

```
jaga x (z : zs)
  = let
      (us , vs) = jaga x zs
    in
      if z <= x then (z : us , vs) else (us , z : vs)
jaga _ _
  = ([] , [])
```

(41)

mis jagab elemendid kaheks listiks vastavalt suvalisele etteantud objektile: viimasest väiksemad või temaga võrdsed paneb esimesse ja suuremad teise listi. Nüüd on sorteerimine kiirmeetodil realiseeritav definitsiooniga

```
qsort (x : xs)
  = let
      (us , vs) = jaga x xs
    in
      qsort us ++ x : qsort vs
qsort _
  = []
```

(42)

Listi kaheks jagamine toimub tema esimese elemendi väärtuse järgi. Miski ei garanteeri, et see on parim valik ega et tulemuseks olev algoritm töötab efektiivselt. See sõltub argumentlistidest, millele funktsiooni `qsort` rakendada tuleb. Juhuslikus järjekorras elementidega listide puhul annab esimese elemendi järgi kaheks grupiks jagamine mõistlikkuse piires erineva pikkusega listid, nii et meetod on kasutatav.

Igal juhul pole definitsioon (42) ka kiirsorteerimisalgoritmi efektiivseim variant, mida Haskellis kirjutada saab. Põhjus on sarnane sellega, miks funktsiooni `reverse` naiivne definitsioon (35) annab ebaefektiivse arvutuse: rekursiivne pöördumine konkatenaatsiooni vasakus argumentis, mille tulemusel tõstetakse juba paika pandud elemente uuesti ümber.

Konkreetselt definitsioonist (42) rääkides siis `qsort us` tulemus, valmissorteeritud listiosa, tõstetakse elementhaaval `x : qsort vs` ette. Kuna `qsort us` ise arvutatakse sama algoritmiga, siis analoogselt osa sellest listist võib järelikult juba tema koostamise käigus varem elementhaaval ümber tõstetud olla, need elemendid siis tõstetakse kokkuvõttes ümber kaks korda. Sama arutelu jätkates saab selgeks, et mida väiksemad on algse listi elemendid, seda rohkem

kordi nad satuvad jaotamisel vasakpoolsesse ossa ning niisama palju kordi tõstetakse neid tühjalt (veel pärast ärasorteerimist) ümber.

Lahendus on sama mis `reverse` puhul — kasutada akumulaatorit, mis võimaldab iga sorteeritud listi elemendid kohe oma õigesse kohta asetada. Otserekursiivse kiirsorteerimise asemel defineerime abifunktsiooni, mis antud listi elemendid sorteerituna akumulaatori ette ühendab:

```
qsort
  = let
    qsort as (x : xs)
      = let
        (us , vs) = jaga x xs
        in
          qsort (x : qsort as vs) us
    qsort as _
      = as
  in
    qsort []
```

(43)

Kuna `us` sorteerimisel käiakse niikuinii tema kõik elemendid läbi, siis keerukusklassi ümber-
tõstmiste kaotamine seekord ei alanda, kuid märgatavalt kiirem on uue definitsiooni järgi töö
sellegipoolest.

Kiirmeetodi vastandina on mestimismeetodi põhimõtteks jagada list kaheks võimalikult kärmelt,
kasutamata ühtki võrdlemist, sorteerida kumbki osa samal meetodil ning mestida tulemused
üheks sorteeritud listiks. Kuna siin puudub võimalus elementide jagamisel n -ö puusse panna,
garanteerib see algoritm töö keerukusklassi $n \log n$ listi pikkuse n suhtes ka halvimal juhul.

Alustame selle meetodi realiseerimist Haskellis viimasest etapist, kahe sorteeritud listi mestimi-
sest. Definitsiooniks võiks olla

```
valitse xs@ (a : as) ys@ (b : bs)
  | a <= b
  = a : valitse as ys
  | otherwise
  = b : valitse xs bs
valitse xs []
  = xs
valitse _ ys
  = ys
```

kus funktsiooni nimi tuleb nüüd sellest, et tegemist on “jaga ja valitse” tehnika rakendamise teise
etapiga.

Kiirmeetodi eeskujul kirjutades saaksime mestimismeetodil sorteerimise defineerida koodiga

```
msort xs@ (_ : _ : _)
  = let
    (us , vs) = split2 xs
    in
      valitse (msort us) (msort vs)
msort xs
  = xs
```

 (44)

kus jagamist teostab koodiga (28) defineeritud funktsioon `split2`. Esimene deklaratsioon definitsioonis (44) sobib juhul, kui listis on vähemalt kaks elementi. Kuna `split2` jagab listi elemendid kaheks võimalikult võrdseks osaks, on mõlemad osad sellisel juhul algsest listist lühemad, nii on rekursiivne pöördumine ohutu. Kui listis on vähem kui kaks elementi, siis pooleks lõõmine ei annaks kaht lühemat listi, seega tuleb teisiti käituda. Kuna 0- ja 1-elementilised listid on juba sorteeritud, piisab anda originaalset välja.

Mestimismeetodil sorteerimine definitsiooniga (44) on keerukusklassis $n \log n$ listi pikkuse n suhtes tänu sellele, et jagamisetapil jaotatakse list alati kaheks enam-vähem võrdseks lühemaks listiks. Kui aga list on juba sorteeritud või vaid mõned elemendid on vales positsioonis, on ka see iteratiivne kaheks jagamine ikka liiga ebaefektiivne, sest sorteerida oleks võimalik ka lineaarse keerukusega.

Üldpõhimõtte poolest sama ideed on võimalik realiseerida ka teisiti, nii et kaheks jagamisel ei puudutataks listi juba sorteeritud osi. Kaheks jagamise puu saab üles ehitada vastupidises järjekorras, lehtedest juure suunas, kui algul teha listist sorteeritud segmentidega listide list ning igal tasemel võtta segmentid kahekaupa kokku ja mestida üheks.

Funktsioon, mis listi järgi leiab tema järjestatud segmentide listi, on juba realiseeritud definitsiooniga (32). Kahekaupa mestimise annab definitsioon

```
kahekaupaMest (xs : ys : yss)
  = valitse xs ys : kahekaupaMest yss
kahekaupaMest xss
  = xss
```

Viimane sooritab ühe taseme üldises sorteerimisprotsessipuu, vähendades segmentide arvu umbes poole võrra. Nüüd on vaja funktsiooni, mis itereeriks kahekaupa mestimist; selle annab definitsioon

```
puuMest xss@ (_ : _ : _)
  = puuMest (kahekaupaMest xss)
puuMest xss
  = xss
```

Selle funktsiooni tulemusel ongi algse listi elemendid sorteeritud, ainult et kui tulemus on mitte-tühi, asub sorteeritud list vastuslistis elemendina. See tuleb sealt lõpuks välja võtta. Seepärast on

sorteerimisfunktsioon nüüd kujul

```
msort xs
  = case puuMest (segmendid xs) of
    zs : _
      -> zs
    _
      -> []
```

(45)

Saadud sorteerimise keerukus on ülimalt $n \log n$ listi pikkuse n suhtes, sest segmente on ülimalt n , igal tasemel väheneb nende arv umbes kaks korda, mistõttu tasemete arv on suurusjärgus $\log n$, ja igal tasemel tuleb kahekaupa mestimisteks teha tööd n sammu. Kui aga list on juba sorteeritud, siis tekib ainult üks segment, üks tase ja jagamisi teha ei tule. Nii on sorteerimine parimal juhul lineaarse keerukusega (ainsa segmendi leidmiseks tuleb list korra läbi vaadata).

Ülesandeid

126. Kirjutada definitsioonide (43) ja (45) eeskujul funktsioonide `qsort` ja `msort` teisendid, mis sorteerides ühtlasi kaotavad elementide kordused, st tulemuslistis esinevad kasvavas järjekorras kõik algse listi elemendid, igauks üks kord.
127. Kirjutada definitsioonide (43) ja (45) eeskujul funktsioonide `qsort` ja `msort` teisendid, mis annavad välja paaride listi, kus paaride esimesed komponendid on parajasti argumentlistis esinevad elemendid, igauks ühe korra, ja vastavad teised komponendid näitavad elemendi esinemise kordsust argumentlistis.
128. Aafrikas elav babababi hõim kasutab tähestikust ainult tähti A ja B. Sõnade moodustamisel kehtivad järgmised ranged reeglid.
1. A on sõna.
 2. Kui u ja v on ühepikkused babababi sõnad, siis $u + v'$ ja $u + v^*$ on babababi 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 asendamisega B-de ja B-de asendamisega A-dega),
 - $u + v$ tähistab sõna, mille saame, kui paaritarvulistele positsioonidele paneme järjekorras u tähed ja paarisarvulistele järjekorras v tähed (flatZip ülesandest 108).
 3. Kõik sõnad on saadavad punktide 1 ja 2 abil.

Kirjutada funktsioon `babababi`, mis kontrollib, kas argumentstring on babababi sõna.

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 saime küll ka juba defineeritud lõpmatutele listidele funktsioone, nagu näiteks konkatenatsioon, `tail` või `drop n`, rakendades, kuid need funktsioonid ei tekita uusi lõpmatuid struktuure, nad on võimelised vaid etteantuid lõplikkuse piires mudima. Lõpmatus pärineb nende funktsioonide puhul alati argumendist.

Lihtsad standardfunktsioonid, mis tõepoolest genereerivad lõpmatuid liste, on `repeat` ja `cycle`. Funktsioon `repeat` suvalisel argumendil x moodustab lõpmatu listi, mille iga element on x . Funktsioon `cycle` võtab argumendiks listi ja annab välja selle listi elementide tsüklilisel kordamisel saadava lõpmatu listi.

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 ilma rekursioonita väljendada ei ole võimalik. Funktsiooni `repeat` võiks defineerida näiteks deklaratsiooniga

```
repeat x
  = x : repeat x
```

(46)

Definitsioon (46) on omapärane selle poolest, et kuigi rekursiivne, ei sisalda ta hargnemiskonstruktsioone, mis seni on ehk paistnud mõistliku rekursiivse definitsiooni juures hädavajalikud, kuna rekursioon peab kuskilt “peale hakkama”. Teisiti öeldult, mõistlikul rekursioonil pidi alati olema baasjuht, kus rekursiivset pöördumist ei toimu, definitsioonil (46) seda aga ei näi olevat, rekursiivne pöördumine toimub iga argumendi puhul.

See rekursioon tõepoolest ei lõpe kunagi ära; kui `repeat x` mingil objektil x väärtustada lasta, tekib lõpmatu arvutus. Kuid niisugune definitsioon on mõttekas tänu sellele, et defineeritava funktsiooni väärtuseks on lõpmatu andmestruktuur, tekkiv lõpmatu arvutus annab pidevalt välja uusi komponente lõpmatust listist. Õigupoolest ei saakski sellisel funktsioonil nagu `repeat` rekursiooni baasjuhtu olla, sest tema argument jääb kogu aeg samaks ja ei kahane protsessi käigus. See argument võib olla mis iganes, ta võib olla objekt mõnest sellisest tüübist, millel kahanemine kui selline pole üldse mõeldavgi, näiteks funktsioonitüübid.

Täpsemalt öeldes on siin tegemist rekursiooni asemel duaalse rekursiooni ehk koorekursiooniga, kus rekursioon toimub mitte argumendi, vaid väärtuse struktuuri järgi. Baasjuhu olemasolu nõudele vastab siin nõue, et igal argumendil peab defineeritav funktsioon varem või hiljem andma välja fragmendi ehitatavast struktuurist; näiteks `repeat x` iga x korral annab enne rekursiivset pöördumist välja listi komponendi x . Kui see tingimus poleks täidetud, tekiks tõepoolest kasutu lõpmatu tsükel.

Analoogselt saab luua ka huvitavamaid liste. Näiteks koodiga

```
jada a d
  = a : jada (a + d) d
```

defineeriksime funktsiooni, mis argumentidel a ja d annab tulemuseks aritmeetilise jada esimese elemendiga a vahega d . Kood

```
suuredSummad
  = let
    suuredSummad a i
      = a : suuredSummad (a + i ^ 4) (i + 1)
    in
    suuredSummad 0 1
```

 (47)

aga defineerib muutuja `suuredSummad` väärtuseks listi, mille elementideks on esimeste naturaalarvude neljandate astmete summad.

Nii saaks eelnevalt juba mitmel viisil defineeritud funktsiooni `suurSumma` defineerida selle kaudu deklaratsiooniga

```
suurSumma n
  | n >= 0
  = suuredSummad !! n
  | otherwise
  = error "suurSumma: negatiivne liidetavate arv"
```

 (48)

Definitsiooni (48) järgi arvutamine on seejuures ainult pisut ebaefektiivsem kui akumulaatoriga definitsiooni (33) järgi arvutamine. Vahe tuleb sellest, et definitsiooni (48) järgi tehakse tekitatakse vahetulemustest list, lisastruktuur, mida endise definitsiooni puhul ei toimu, ning lõpus lisandub listist lugemine. Õigupoolest ongi definitsioon (47) väga sarnane akumulaatoriga definitsiooniga (33). Lokaalse abifunktsiooni parameeter a on siingi akumuleeruv ja i tsükliindeks. Siin lihtsalt moodustatakse akumulaatori väärtustest ühtlasi list. Teine erinevus on rekursiooni baasjuhu puudumine.

Seega peaaegu sama hinnaga kui enne `suurSumma n` väärtuse mingi konkreetse n jaoks saame nüüd kätte kõik väärtused `suurSumma i`, kus $0 \leq i \leq n$. Definitsiooni (47) sobivasse kohta agara väärtustamise operaatori lisamisel saame ka mälu kasutuse praktiliselt samale tasemele viia kui definitsiooni (34) järgi arvutamisel, kus akumulaatoris toimub jooksev väärtustamine.

Kokkuvõttes kui on mingi arvutuse käigus vaja `suurSumma` väärtusi paljudel argumentidel, tasub eelistada nende listi moodustamist.

Ülesandeid

129. Testida interaktiivse interpretaatori käsurealt senivaadeldud viisidel konstrueeritavaid lõpmatuid liste.

130. Kirjutada definitsiooni (47) teisend, mille puhul definitsiooni (48) järgi arvutamisel toimuks vahesummade jooksev väärtustamine.
131. Defineerida muutuja `facts` väärtuseks list, mille väärtuseks on järjest kõigi naturaalarvude faktoriaalid. Jälgida, et listi algusosa väljaarvutamine oleks lineaarse keerukusega arvutatud osa pikkuse suhtes.
132. Olgu `,` paremassotsiatiivne binaarne operatsioon, mis defineeritakse reeglina

$$a , r = a + \frac{1}{r}.$$

Reaalarvu r ahelmurruks (ingl *continued fraction*) nimetatakse lõplikku või lõpmatut esitust vastavalt kujul $r = a_0 , \dots , a_n$ või $r = a_0 , a_1 , \dots$, 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 funktsioon `ahel`, mis ratsionaalarvul r arvutab r ahelmurru. Ahelmurdu kujutada ahelmurru elementide listina.

Leida selle definitsiooni sarnasusi ja erinevusi võrreldes teiste selles jaotises käsitletutega.

Seni defineerisime lõpmatuid liste küll rekursiooni abil, kuid rekursiivselt oli defineeritud ikkagi funktsioon, mitte list. Näiteks definitsioon (46) defineerib rekursiivselt funktsiooni `repeat`, mitte mõnd tema väärtustest, sest rekursiooni tekitab paremas pooles esinev funktsiooni `repeat` väljakutse.

Tänu laisale väärtustamisele on Haskellis võimalik andmestruktuure ka otse rekursiivselt defineerida. Definitsiooni (46) asemel võiksime kirjutada hoopis

```
repeat x
  = let
      xs = x : xs .
    in
      xs
```

(49)

Funktsiooni `repeat` rekursiivsest väljakutsest oleme lahti saanud. Selle asemel kasutab lokaalse muutuja `xs` definitsiooni parem pool sedasama defineeritavat listi `xs`. Seega list defineeritakse rekursiivselt. Tulemus on sama mis enne.

Ülesandeid

133. Leida moodulist Prelude funktsiooni `cycle` definitsioon ja saada sellest aru.

Listi defineerimisel rekursiivselt tuleb mõelda, mis on listi pea ja kuidas listi saba avaldub terve listi kaudu — või, keerulisemal juhul, mis on listi algusjupp kuni mingi kindla kohani ja kuidas

listi ülejääv osa alates mingist elemendist avaldub kogu listi kaudu. Näiteks `repeat x` puhul on peaks x ja saba võrdub listi endaga. Siit ka seos $xs = x : xs$ definitsioonis (49).

Ka funktsiooni `jada` saab defineerida nii, et rekursioon toimub listil. Selleks paneme tähele, et mistahes a ja d korral listi `jada a d` pea on a ja saba saame kogu listist, kui tema igale elemendile liidame d . Definitsiooniks tuleb

```
jada a d
= let
  as = a : [x + d | x <- as] .
  in
  as
```

Takistusi pole veelgi keerulisemaks minekul. Oletame, et tahame kirjutada 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 ning panna nende kahe listi elemendid kasvavas järjekorras ühte kokku, kaotades ka kordused. Kahe sorteeritud listi elementide kasvavas järjekorras kokkupanekut oleme käsitlenud, see on mestimine. Pole raske seda modifitseerida nii, et kahe kordumisteta sorteeritud listi puhul tekiks kordumisteta sorteeritud list. Kuna terve list on kordumisteta ja sorteeritud, siis ka 2-kordsete ja 5-kordsete list on kumbki kordumisteta ja sorteeritud, seega selline mestimine kõlbabki. Tulemuseks on definitsioonid

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

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

Defineerime sama tehnikaga ka listi, mille elementideks on järjekorras kõik Fibonacci arvud. Selleks sobib definitsioon

```
fibs = 0 : 1 : summad fibs, (50)
```

kus funktsioon `summad` on antud definitsiooniga

```
summad (x : xs)
  = let
      summad x (z : zs)
        = x + z : summad z zs
      summad _ _
        = []
    in
      summad x xs
summad _
  = [] (51)
```

ta leiab listi järjestikuste liikmete summade listi. Kuna iga Fibonacci arv peale esimese kahe võrdub kahe eelmise Fibonacci arvu summaga, siis `summad fibs`, kui `fibs` on list järjekorras kõigi Fibonacci arvudega, on parajasti `fibs` ilma kahe esimese elemendita. Siit ka definitsioon (50).

Definitsiooni (50) järgi toimub kõigi Fibonacci arvude kuni n -ndani väljaarvutamine kokku lineaarse keerukusega n suhtes. Eelmistes jaotistes korduvalt defineeritud funktsioonile `fib` võib anda niisiis veel ühe lineaarselt töötava definitsiooni

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

Ainus puudus on see, et arvutuses tekkivaid summasid ei väärtustata jooksvalt.

Definitsiooni (51) teist deklaratsiooni Fibonacci arvude listi arvutamine ei vaja, sest list, millele see funktsioon on rakendatud, ei lõpe kunagi ära. See deklaratsioon on pandud täielikkuse huvides, juhuks kui keegi tahab teda väikestel listidel testida või muidu kasutada.

Ülesandeid

134. Modifitseerida definitsiooni (51) selliselt, et `fibs` arvutamisel toimuks jooksev summade arvutamine.

135. Defineerida rekursiivselt muutuja `hamming` väärtuseks list, mille elementideks on kasvavas järjekorras kõik positiivsed täisarvud, mille kanoonilises esituses ei esine muud algarvud peale 2, 3, 5.
136. 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 kahekordselt jne.
137. Defineerida rekursiivselt muutuja `bitistringid` väärtuseks list, mille elementideks on kõik lõplikud bitistringid.
138. Defineerida rekursiivselt muutuja `paarid` väärtuseks list, mille elementideks on kõik naturaalarvupaarid.
139. Defineerida muutuja, mille väärtustamisel tekib lõpmata palju veateateid.

Rekursiivselt defineeritud andmestruktuur ei pruugi alati lõpmatu olla ega isegi mitte väljaarvutamisel lõpmatut protsessi tekitada. Näiteks koodiga

```

blah n x
  = let
      xs = x : take n xs
    in
      xs

```

defineeritud listid `blah n x` koosnevad parajasti $n + 1$ elemendist x . On võimalikud igasugused veidrad rekursiivsed andmestruktuuridefinitsioonid, mis lõpetavad paari sammuga, näiteks definitsioon

$$p = (9, \text{fst } p - 1) \tag{52}$$

defineerib `p` väärtuseks paari (8, 9).

Ülesandeid

140. Vaatame deklaratsiooniga (52) analoogset deklaratsiooni

$$q = (\text{snd } q - 1, 9)$$

Mis saab muutuja `q` väärtuseks?

Kõrgemat järku funktsioonid

Eelnevas oleme näinud, et Haskell lubab funktsioonil võtta teist funktsiooni argumentiks. Funktsioone argumentiks võtvaid funktsioone kutsutakse kõrgemat järku funktsioonideks. Funktsionaalses programmeerimises on kõrgemat järku funktsioonid üldlevinud nähtuseks.

Kuna funktsiooniga programmeeritakse üldiselt teatavat käitumist, võimaldab funktsioon funktsiooni argumentina esitada teatavat käitumist parameetrisena teise käitumise suhtes. Niisuguselt defineeritud funktsioonid on laiemas kasutusega kui funktsioonid, mille argumentide hulgas funktsioone pole, ja võimaldavad vältida koodikordust (enam-vähem sama käitumise programmeerimist ikka ja jälle uuesti).

Haskellis on kõrgemat järku funktsioonid enamasti ühtlasi polümorfsed, mis tähendab, et nad on kasutatavad paljude erinevate tüüpide jaoks, omavad lisaks väärtusparameetritele ka implitsiitseid tüübiparameetreid (implitsiitseid seetõttu, et funktsiooni definitsioonis nad ei kajastu, ainult signatuuris on nad ilmutatud). See teeb need funktsioonid veelgi universaalsemaks.

Paljud moodulis Prelude ja mujal standarteegis eeldefineeritud funktsioonid on mõeldud selliseks universaalseks kasutamiseks. Kui konkreetne ülesanne või alamülesanne lahendub mõne universaalse funktsiooni rakendusena, on halb stiil minna sellest võimalusest mööda. Universaalsete funktsioonide õige kasutamine tagab koodi lugejale kiirema arusaamise programmist.

Prelüüdi funktsioon `flip` võtab argumentiks *curried*-kujul funktsiooni f ja annab tulemuseks funktsiooni, mis töötab nagu f , kuid võtab kaks esimest argumenti vastupidises järjekorras. Näiteks kui on vaja defineerida funktsioon, mis töötab nagu täisarvuline jagamine, kuid võtab argumentideks jagaja enne jagatavat, on seda kõige mugavam ja kergemini loetavam teha funktsiooniga `flip`:

```
minuJagamine = flip div. (53)
```

Ülesandeid

141. Testida interaktiivse interpretaatori käsurealt funktsiooni `flip`, sealhulgas argumentiga `div`.
142. Kirjutada oma moodulisse definitsioon (53) ja testida interaktiivse interpretaatori käsurealt funktsiooni `minuJagamine`.
143. Defineerida võimalikult lühidalt *curried*-kujul funktsioon `minuLahutamine`, mis võtab järjest kaks argumenti ja annab välja teise argumenti ja esimese argumenti vahe.
144. Defineerda võimalikult lühidalt *curried*-kujul funktsioon `vahetaInfiks`, mis võtab järjest kolm argumenti x , \oplus , y , millest teine on infiksoperaator, ja annab neil tulemuseks

objekti $y \oplus x$. (See on infiksoperaatori argumentide vahetamine, kus infiksoperaator jääb oma seniste argumentide vahele.)

Kui funktsiooni `flip` argumentfunktsiooni üks argument on antud, võib `flip` asemel kasutada paremseltsiooni. (Kompilaator tegelikult kirjutabki paremseltsioonid ümber funktsiooni `flip` kaudu, tuum-Haskellis seltsioone ei ole.) Näiteks `flip div 2` on funktsioon, mis jagab oma argumenti täisarvuliselt 2-ga, st sama mis `(`div` 2)`.

Kui funktsiooni käitumine seisneb oma argumendile mingite funktsioonide järjest rakendamises, kusjuures need funktsioonid sellest argumendist ei sõltu, võib kasutada kompositsiooni. Kompositsioon on Haskellis infiksoperaatori `.` väärtuseks. Tegemist on kõrgemat järku funktsiooniga, kuna ta võtab funktsioone (koguni kaks) argumendiks.

Sobivate tüüpidega funktsioonide f ja g korral on $f \cdot g$ väärtuseks funktsioon, mis rakendab oma argumendile kõigepealt funktsiooni g ja saadud tulemusele seejärel funktsiooni f . Näiteks avaldise `(+ 2) . (* 5)` väärtuseks on funktsioon, mis igal oma argumendil arvutab väärtuse, korrutades argumenti 5-ga ja liites saadud tulemusele 2.

Prelüüdi funktsioonid `curry` ja `uncurry` teisendavad binaarsete funktsioonide *curried*-kuju ja korteežiargumendiga kuju vahel. Eelnevas nägime, et näiteks funktsioonile `fst` vastav *curried*-kujul funktsioon on `const`. Seega funktsioon `curry fst` käitub nagu `const`, funktsioon `uncurry const` aga nagu `fst`.

Ülesandeid

145. Kirjutada avaldis, mille väärtuseks on funktsioon, mis arvutab väärtusi, võttes oma argumentid siinuse ja saadud tulemusest koosinuse.
146. Veenduda interaktiivse interpretaatori käsurealt testides, et `curry fst` töötab nagu `const` ja `uncurry const` töötab nagu `fst`.
147. Kirjutada avaldis, mille väärtuseks on *curried*-kujul funktsioon, mis võtab järjest kaks argumenti ja annab välja neist teise.

Loengus on seletatud paljusid muid prelüüdi kõrgemat järku polümorfseid funktsioone — enamasti listifunktsioone, nagu näiteks `map`, `filter`, `takeWhile`, `dropWhile`, `foldr`, `foldl`, `scanr`, `scanl`, `zipWith`. Lisada võiks veel funktsioonid `all` ja `any`, mis mõlemad võtavad argumendiks predikaadi (nagu `filter`) ja annavad väärtuseks funktsiooni, mis võtab argumendiks listi ja annab välja tõeväärtuse; funktsiooni `all` puhul ütleb see tõeväärtus, kas listi kõik elemendid rahuldavad predikaati, funktsiooni `any` puhul aga, kas listis leidub predikaati rahuldav element.

Kui vaja on näiteks funktsiooni, mis võtab argumendiks listi ja annab tulemuseks sellest listist kõigi elementide kahekordistamise teel saadud listi, ei peaks mitte rekursiooniskeemi program-

meerima hakkama, vaid kasutama standardfunktsiooni `map`. Definiitsioon tuleks

```
double = map (* 2).
```

Nii lihtsa definiitsiooni puhul tasuks kaaluda lausa mõtet uue muutuja `double` kasutuselevõttust loobuda ja, igal pool kus vaja, kirjutada lihtsalt `map (* 2)` sisse, sest efektiivsusevõitu siin muutuja kasutuselevõttust ei järgne.

Ülesandeid

148. Väärtustada interaktiivse interpretaatori käsurealt mõni avaldis, kus rakendatakse funktsiooni `map`.
149. Kirjutada avaldis, mille väärtus on funktsioon, mis argumentstringil annab välja järjekorras selle stringi kõigi sümbolite koodid listina.
150. Arvutada string koodide järjekorras kõigist sümbolitest, mis kooditabelis leiduvad.
151. Defineerida funktsioon `ilmaÜhetaSummad`, mis võtab argumentiks listi l ja annab välja listi, milles on samapalju elemente ja mille element positsioonil i võrdub summaga listi l kõigist elementidest peale selle, mis on positsioonil i .
152. Defineerida funktsioon `nihet`, mis täisarvulisel argumentil n annab nihkešifri n sümboli võrra. Nihkešifri n sümboli võrra on funktsioon, mis argumentstringil s annab välja stringi, mille saab, kui s iga tähe kohale paneb temast ladina tähestikus n sümboli võrra tagapool asuva tähe, interpreteerides tähestikku tsüklilisena (st “z” järel tuleb jälle “a”). Argumentstringi väiketähtede kohal peavad olema väiketähed, suurtähtede kohal suurtähed.
153. Kirjutada funktsioon `vahedeKorrutis`, mis võtab argumentiks arvude listi ja annab väärtuseks tema mistahes kahest erinevast kohast võetud elementide vahede (eespoolsest lahutatakse tagapoolne) korrutise.
154. Defineerida muutuja, mille väärtuseks on lõpmatu list lõpmatutest listidest, ja rakendada interaktiivse interpretaatori käsurealt talle selline funktsioon, mis leiab neist igauhest mõned elemendid.
155. Defineerida muutuja `ükskordüks`, mille väärtuseks on lõpmatu naturaalarvude korrutustabel listide listina, iga list selles listis väljendab tabeli üht rida. Testida seda muutujat interaktiivse interpretaatori käsurealt, lastes väärtustada tabeli ülemise 10×10 nurga.

Mõnikord pole funktsiooni `map` rakendamise tulemus otseselt see, mida ülesanne nõudis, nii et tulemust peab veel töötlema. See pole argument funktsiooni `map` kasutamisest loobumise kasuks. Tihti juhtub, et listi iga elemendiga on põhimõtteliselt vaja sooritada operatsioon f , mille tulemus on ise list, kuid tulemus ei tohi olla listide list, soovitud tulemuse aga saame sellest funktsiooni `concat` rakendamisega.

Olgu vaja kirjutada funktsioon `asendaTab`, mis asendab stringis tabulaatorid kaheksakordse tühikuga, jättes muud sümbolid paigale. Sellise funktsiooni tüüp oleks `String -> String`, mis on sama mis `[Char] -> [Char]`. Kui `asendaTab` esituks kujul `map f`, siis peaks `f` olema tüüpi `Char -> Char`. Kuna aga kaheksakordne tühik on mitte sümbol, vaid string, siis pole võimalik vajalikku asendusreeglit niisuguse funktsioonina esitada, järelikult funktsioon `asendaTab` ei esitu kujul `map f`. Küll aga saame soovitud tulemuse, kui me pärast `map f` rakendamist lubame kasutada funktsiooni `concat`. Õige definitsioon oleks

```
asendaTab xs
  = concat (
    map (\ x -> if x == '\t' then "      " else [x]) xs (54)
  )
```

Konstruksioon, kus funktsioonile kujul `map f` järgneb `concat`, on nii levinud, et prelüüdis on defineeritud spetsiaalne muutuja `concatMap`, mille väärtuseks on *curried*-kujul funktsioon, mis võtab argumendiks funktsiooni `f` ja listi `l` ning arvutab tulemuse, rakendades listile `l` funktsiooni `map f` ja saadud listile funktsiooni `concat`.

Vähe sellest, prelüüd defineerib ka infiksoperaatorid `>>=` ja `=<<`, mis kumbki teeb sisuliselt sedasama, ainult `>>=` võtab seejuures argumendid vastupidises järjekorras. Niisiis `>>=` väärtus on sama mis avaldisel `flip concatMap`. Operaatori `>>=` abil saaksime definitsiooniga (54) antud funktsiooni `asendaTab` defineerida kujul

```
asendaTab = (>>= \ x -> if x == '\t' then "      "      " else [x]).
```

Ülesandeid

156. Anda funktsioonile `asendaTab` definitsiooni (54) asemel selline definitsioon, kus kasutatakse operaatorit `=<<`.
157. Kirjutada `>>=` abil funktsioon `toDOS`, mis asendab argumentstringis esinevad sümbolid “\n” (reavahetused) DOSi kodeeringu reavahetustega, milleks on kahesümboliline järjend “\r\n”. (Haskellis süntaksis on sümbolite kirjakujud “\n” ja “\r” olemas.)
158. Kirjutada *curried*-kujul funktsioon `venita`, mis võtab ükshaaval argumendiks arvu n ja listi l ja annab tulemuseks listi, mille saab listist l elementide n -kordsel kordamisel.

Operaator `>>=` aitab ka juhul, kui mõni element tuleb listis ära kaotada. Näiteks kui p on predikaat, siis funktsioon, mis argumentlistil l annab välja listi, mille ta saab listist l predikaati p mitte rahuldavate elementide väljajätmisel, on

```
(>>= \ x -> if p x then [x] else []), (55)
```

kus p on avaldis, mille väärtuseks on p . Niisugust operatsiooni saab aga paremini teha kõrgemat järku funktsiooniga `filter`; avaldisega (55) samaväärne on avaldis `filter p`.

Ülesandeid

159. Kirjutada avaldis, mille väärtuseks on list paaritute binoomkordajatest $\binom{n}{2}$.

160. Kirjutada funktsioon, mis argumenttekstil annab välja tema pikima väiketähti mitte sisaldava algusjupi.

161. Lahendada ülesanne 80 komprehensioonsüntaksit kasutamata.

Funktsioon `foldr` võtab argumentideks järjest operatsiooni \oplus , objekti e ja listi l ning annab välja objekti, mille saab, kui listi elemendid järjekorras operatsiooniga \oplus kokku arvutab, asetades sulud paremalt ja võttes algelemendiks e . Kui l elemendid järjekorras on a_1, \dots, a_n , siis tulemuseks on $a_1 \oplus (a_2 \oplus \dots \oplus (a_n \oplus e))$. Erijuhul, kui list on tühi, on tulemuseks lihtsalt e .

Kui programmeerimisel mõni alamülesanne on esitatav `foldr` kaudu, on tavaliselt soovitatav ta nii ka esitada, sest niisugune listi elementide kokkuarvutamisskeem on üks põhilisi ja selle ilmutatud väljatoomine annab lugejale kiiresti selguse funktsiooni olemusest. Selgitustest eelmises lõigus nähtub, et kujul `foldr \oplus e` esituvad ainult sellised funktsioonid, mille käitumine mittetühjal listil on määratud listi pea ja sama funktsiooni töö tulemusega listi sabal.

Loengumaterjalis on toodud palju näiteid funktsioonide definitsioonidest kujul `foldr \oplus e`. Funktsiooni f esitamiseks kujul `foldr \oplus e` tuleb e ja \oplus leida nii, et e oleks funktsiooni f väärtus tühjal listil ning \oplus oleks selline operaator, et iga mittetühja listi l korral, rakendades \oplus listi l peale ja funktsiooni f väärtusele listi l sabal, oleks tulemuseks f väärtus kogu listil l .

Esitame sellisel kujul näiteks funktsiooni `split2`, mis on antud definitsiooniga (28) ja mis jaotab oma argumentlisti elemendid üle ühe kahte listi. Algvärtus e peab olema $([], [])$, selle saab otse definitsioonist (28) välja lugeda. Operaator \oplus peab võtma listi pea ja funktsiooni väärtuse listi sabal, mis kujutab endast paari kahest listist, ja andma välja funktsiooni väärtuse kogu listil. Selle põhjal saame definitsiooni

$$\begin{aligned} \text{split2} \\ = \text{foldr } (\backslash x \text{ (us , vs) } \rightarrow (x : \text{vs} , \text{us})) ([] , []). \end{aligned} \quad (56)$$

Definitsioon (56) ei anna siiski alati samu väljundeid nagu definitsioon (28). Vahe tekib lõpmatutel listidel, kus definitsiooniga (56) antud funktsioon jääb lõpmatusse tsüklisse ilma mingigi väljundita, definitsiooniga (28) antud funktsioon aga annab paari kahest lõpmatust listist.

Esmapilgul võib tunduda, et funktsioonid kujul `foldr \oplus e` ei saa lõpmatule listile rakendatult kunagi midagi mõistlikku anda, sest nende väärtus lõpmatul listil on antud avaldisega

$$a_0 \oplus (a_1 \oplus (a_2 \oplus (\dots))),$$

milles pole ühtki lõplikku alamavaldist, kust ehk arvutus peale saaks hakata. See mulje on aga kahel põhjusel ekslik. Esiteks võib operaator \oplus olla oma parema argumenti järgi laisk ja nii-pea, kui mõne a_i korral \oplus ei vaja väärtuse leidmiseks teist argumenti, unustatakse see ära ja

järele jääb juba lõplik avaldis. Niisiis on võimalik, et `foldr ⊕ e` lõpetab lõpmatul listil töö normaalselt lõpliku ajaga. Teiseks võib niisuguse avaldise väärtus olla lõpmatu andmestruktuur, millest operatori \oplus iga rakendus annab juba ainuüksi oma vasakpoolse argumendi põhjal mõne komponendi kätte.

Funktsiooni `split2` puhul on vaja just nimelt koostada lõpmatust sisaldavat andmestruktuuri — paari kahe lõpmatu listiga — ja peaks olema võimalik `foldr` niimoodi tööle panna, et ta järk-järgult nende komponente välja arvutaks. Definiitsioon (56) ei tee seda sellepärast, et `foldr` argumentoperaatori teise argumendi näidis (`us`, `vs`) on agar, nõudes argumendi sobitamist temaga enne operatori rakendamist. Et teine argument on rekursiivse pöördumise tulemus, siis ei saa operatsiooni rakendada enne, kui rekursiivne pöördumine listi sabale on paari välja andnud — see aga ei juhtu enne, kui ta on rakendanud operatsiooni. See nõiaring osutab, et esimene paar genereeritakse alles siis, kui rekursioon on jõudnud põhjani, tühja listini. Lõpmatu listi puhul aga ei juhtu seda kunagi.

Olukorra parandamiseks tuleb paarinäidis laisaks sundida, et oleks võimalik operatsiooni täitmisega alustada enne rekursiivse pöördumise lõpetamist. Saame definiitsiooni

```
split2
  = foldr (\ x ~(us , vs) -> (x : vs , us)) ([] , [])'      (57)
```

mis töötab soovitud viisil ka lõpmatutel listidel. See ongi laisaks sundimise tüüpilisim praktiline kasutus.

Veel näitame, kuidas esitada kujul `foldr ⊕ e` funktsioon `eraldaSegment`, mis on antud definiitsiooniga (31) ja mis argumentlistil annab välja listipaari, kus esimeses listis on argumentlisti pikim mittekahanev aligusjupp ja teises listis kõik ülejäänud elemendid. Eelmise näite eeskujul tehes on tulemuseks definiitsioon

```
eraldaSegment
  = foldr (
    \ x ~(us@ ~(y : _) , vs)
      -> if null us || x > y
          then ([x] , us ++ vs)
          else (x : us , vs)
    )
    ([] , [])      (58)
```

Vastavalt definiitsioonile (31) tuleb tühjal listil väärtuseks anda `([], [])`, sellest ka `foldr` teine argument `([], [])`. Kui mittetühja listi pea on `x` ja sabal annab `eraldaSegment` välja paari (k, l) , siis on kaks võimalust. Kui `k` on tühi või tema esimene element on `x`-st väiksem, siis esimene segment ainult `x`-st koosnebki ja väljaantava paari teine komponent koosneb kõigist ülejäänud elementidest, mille saame kätte `k` ja `l` konkateneerimisel, sest (k, l) on saba elementide jaotus esimeseks segmendiks ja ülejäänud osaks. Vastasel korral, kui `k` on mittetühi ja tema esimene element pole `x`-st väiksem, moodustab `x` koos `k` elementidega ühise esimese segmendi kogu listi jaoks.

Kõik paistab klappivat, ka paarinäidis on laisaks sunnitud. Ometi see definitsioon lõpmatul listil ei tööta, sest kuigi paarinäidist ei sobitata enne operatsiooni alustamist, nõutakse selles näidised defineeritud muutujaid us ja y kohe operatsiooni algul kontrollis, nii et sisulist vahet agaraks jäetud näidisega pole. Definitsioon (57) töötab lõpmatul listil tänu sellele, et operatsioon moodustab paari ja paneb tema esimese komponentlisti esimese elemendi paika enne oma argumentpaari uurimist. Definitsioonis (58) aga seda ei tehta.

Olukorra päästmiseks tuleb operatsiooni käitumine ümber defineerida, et ta paneks ühe elemendi paika enne argumentpaari uurimist. Definitsiooni (58) uurimine näitab, et igal juhul peaks listi pea minema tulemuspaari esimese komponentlisti peaks. Toome selle tegevuse tingimusavaldise seest välja. Nii saame korralikult töötava definitsiooni

```

eraldaSegment
  = foldr (
    \ x p@ ~(us@ ~(y : _) , vs)
      -> let
          (us' , vs') = if null us || x > y
                        then ([] , us ++ vs) .
                        else p
        in
          (x : us' , vs')
    )
    ([] , [])

```

(59)

Suure töö tulemusena saadud definitsiooni (59) nõrgaks küljeks on aga, et temaga arvutamine on tunduvalt ebaefektiivsem kui algse definitsiooniga (31), kuna mitte ainult esimene segment, vaid kogu listi elemendid tõstetakse segmenthaaval ümber ja järgnevad segmendid konkateneeritakse uuesti üksteise külge, mis on tühi töö.

Samas leidub palju listifunktsioone, mis kujul $\text{foldr } \oplus e$ ei esitu. Selline on näiteks definitsiooniga (29) või (30) antav funktsioon `kõikVõrdsed`, mis etteantud listi järgi otsutab, kas listi elemendid on kõik võrdsed, kuna vajab lisaks rekursiivse töö tulemusele listi sabal — definitsioonis (30) kutse `kõikVõrdsed xs` — ka listi saba esimest elementi — y . Samale järeldusele jõuame ka ilma definitsioone vaatamata, uurides paljalt funktsiooni `kõikVõrdsed` tähendust: seda, kas mittetühja listi kõik elemendid on võrdsed, pole võimalik määrata paljalt listi pea ja selle baasil, kas listi saba kõik elemendid on võrdsed, sest see info kokku ei näita, kas listi pea on võrdne listi saba kõigi elementidega või on ta neist erinev.

Ülesandeid

162. Väljendada kujul $\text{foldr } \oplus e$ kõik standardfunktsioonid, mis võimalik, järgmisest loetelust:

- head;

- tail;
- null;
- take n ;
- drop n .

Nõutud kujul väljenduvate funktsioonide korral teha kindlaks, kas kirjutatud definitsioon `foldr` kaudu on niisama efektiivne kui varasem definitsioon.

163. Väljendada kujul `foldr ⊕ e` kõik funktsioonid, mis võimalik, järgmisest loetelust:

- ($>>= k$);
- filter p ;
- takeWhile p ;
- dropWhile p .

Nõutud kujul väljenduvate funktsioonide korral teha kindlaks, kas kirjutatud definitsioon `foldr` kaudu on niisama efektiivne kui varasem definitsioon.

164. Väljendada kujul `foldr ⊕ e` kõik praktikummaterjalis esinevad funktsioonid, mis võimalik, järgmisest loetelust:

- eemaldaEsimene (definitsioon (27));
- eemaldaKõik (ülesanne 107);
- segmendid (definitsioon (32));
- korduvad (ülesanne 114);
- summad (definitsioon (51));
- ilmaÜhetaSummad (ülesanne (151));
- vahedeKorrutis (ülesanne (153)).

Nõutud kujul väljenduvate funktsioonide korral teha kindlaks, kas kirjutatud definitsioon `foldr` kaudu on niisama efektiivne kui varasem definitsioon.

165. Leida selline operaator \oplus ja väärtus e , et mingi lõpmatu listi l korral lõpetab `foldr ⊕ e l` arvutus töö normaalselt lõpliku ajaga, kuid iga lõpmatu listi l korral jääb `foldr (flip ⊕) e l` lõpmatusse tsüklisse ilma midagi välja andmata.

Ka funktsiooni `foldl` kasutamisest on loengumaterjalis mitu näidet. Nagu `foldr`, võtab ka `foldl` argumentideks järjest operatsiooni \oplus , objekti e ja listi l ning annab välja objekti, mille saab, kui listi elemendid järjekorras operatsiooniga \oplus kokku arvutab, võttes algelemendiks e , kuid `foldl` asetab sulud vasakult. Kui l elemendid järjekorras on a_1, \dots, a_n , siis tulemuseks on $(\dots((e \oplus a_1) \oplus a_2) \oplus \dots) \oplus a_n$. Kui list on tühi, on tulemuseks lihtsalt e .

Funktsioon `foldl` abstraherib kõiki selliseid ühe akumulaatoriga arvutamisi, kus akumulaatori uus väärtus leitakse vanast, rakendades talle ja mingile elemendile antud operatsiooni (`foldl` argumenti), ja lõpus antakse akumulaator välja. Akumulaatoriga arvutamise esitamiseks funktsiooni `foldl` abil tuleb need elemendid, millega akumulaatorit kokku opereeritakse, paigutada järjekorras listi, mis antakse funktsioonile `foldl` viimaseks argumentiks. Teiseks argumentiks `e` on akumulaatori algväärtus.

Näiteks funktsiooni `suurSumma` arvutamine akumulaatoriga, mille realiseerib definitsioon (33), on abstraheritav funktsiooni `foldl` abil. Uueks definitsiooniks saame

```
suurSumma n
  | n >= 0
  = foldl (\ a i -> a + i ^ 4) 0 [1 .. n]
  | otherwise
  = error "suurSumma: negatiivne liidetavate arv" (60)
```

Siin `foldl` argumentoperatsioon näitab, et akumulaatori uus väärtus saadakse jooksvast, liites talle listi jooksva elemendi neljanda astme, teine argument 0 näitab, et akumulaatori algväärtus on 0. Kuna listiarargument koosneb elementidest 1 kuni n , siis antud funktsioon kokkuvõttes liidab arvule 0 järjest arvude 1 kuni n neljandad astmed.

Funktsiooni `foldl` kasutamine muudab küll koodi lühemaks ja loetavamaks, kuid temaga kaasneb oluline puudus, et akumulaatorit ei väärtustata jooksvalt ja selle heaks pole ka võimalik midagi ette võtta.

Veel tuleb arvestada, et lõpmatu listi puhul viib funktsioon kujul `foldl e` väärtustamisel alati lõpmatusse tsüklisse ilma midagi välja andmata.

Ülesandeid

166. Võidakse arvata, et funktsiooni `suurSumma` saab sundida akumulaatorit agaralt väärtustama, kirjutades definitsiooni (60) ümber kujul

```
suurSumma n
  | n >= 0
  = foldl (($!) (\ a i -> a + i ^ 4)) 0 [1 .. n]
  | otherwise
  = error "suurSumma: negatiivne liidetavate arv"
```

Miks see arvamus ei ole õige?

167. Defineerida funktsiooni `foldl` abil *curried*-kujul funktsioon `kfact`, mille väärtus argumentidel n ja n on $(n)_k$ (n kahanev faktoriaal k järgi, defineeritud ülesandes 118).

168. Defineerida kujul $\text{foldl } \oplus e$ funktsioon `loemÄrgid`, mis argumentlistil annab välja kolmiku, mille komponendid on järjest argumentlisti negatiivsete komponentide, nullide ja positiivsete komponentide arv.

Akumulaatori vaheväärtusi listi salvestava operatsiooni abstraktsioon on kõrgemat järku funktsioon `scanl`. Näiteks definitsiooniga (47) antud funktsiooni saab samaväärselt defineerida kujul

```
suuredSummad = scanl (\ a i -> a + i ^ 4) 0 [1 .. ].
```

Ülesandeid

169. Lahendada ülesanne 131, kasutades funktsiooni `scanl`.

Liste genereerib ka kõrgemat järku funktsioon `iterate`, mis võtab järjest argumentideks mingist tüübist samasse tüüpi töötava funktsiooni f ja tema võimaliku argumenti x ning annab väärtuseks lõpmatu listi elementidega $x, f x, f (f x), f (f (f x))$ jne. Näiteks avaldise `iterate (* 2) 1` väärtuseks on lõpmatu list elementidega $1, 2, 4, 8, 16, \dots$

Mõneti sarnane, samuti iteratiivset käitumist abstraheriv, on kõrgemat järku funktsioon `until`. Tema võtab argumentideks järjest predikaadi p , tema argumenttüübist samasse tüüpi töötava funktsiooni f ning argumenti x ning annab välja esimese väärtuse jadas $x, f x, f (f x)$ jne, mis rahuldab predikaati p , kui selline leidub, vastasel korral jääb lõpmatusse tsükklisse. Näiteks avaldise `until (> 100) (* 2) 1` väärtus on 128, sest 128 on esimene arvu 2 aste, mis on suurem 100-st.

Kõrgemat järku funktsiooni `zipWith` abil saab elegantselt realiseerida näiteks funktsioone, mis nõuavad listi kahe järjestikuse elemendi üheaegset vaatlemist igal rekursioonisammul. Oleme selliseid seni mitmeid defineerinud — definitsioonid on olnud suhteliselt kohmakad, näiteks funktsiooni `kõikVõrdsed` definitsioonid (29) ja (30) ja funktsiooni `summad` definitsioon (51). Funktsiooniga `zipWith` on näiteks funktsioon `summad` defineeritav koodiga

```
summad xs
  = zipWith (+) xs (tail xs) (61)
```

Siin `tail xs` tekitab argumentlisti nihke ühe komponendi võrra, nii et vastavaid elemente kokku liitev `zipWith (+)` saab igal sammul parajasti algse listi kaks järjestikust elementi ette.

Pangem tähele, et definitsioon (61) töötab ka juhul, kui argumentlist `xs` on tühi ja `tail xs` seega vigane. Põhjus peitub selles, et `zipWith` uurib liste ainult niikaua, kui üks neist ära lõpeb; kui esimene list on tühi, siis antakse tühi list välja ja teine list jääb üldse uurimata.

Funktsiooni `summad` läks vaja Fibonacci jada arvutamiseks rekursiivse listi tehnika abil definitsiooniga (50). Asendades seal pöördumise funktsiooni `summad` poole definitsiooni (61) parema poole järgi, saame variandi ilma `summad` poole pöördumata:

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs).
```

Pisut teisendades saame elegantsema definitsiooni

```
fibs@ (_ : fs) = 0 : 1 : zipWith (+) fibs fs.
```

Kuid tõenäoliselt kõige elegantsem definitsioon Fibonacci arvude listile on siiski funktsiooni `scanl` abil:

```
fibs = 0 : scanl (+) 1 fibs.
```

Ülesandeid

170. Kui `zipWith` argumentoperaator on kommutatiivne, kas siis teeb alati sama välja, ükskõik kummas järjekorras anda listid?
171. Kirjutada interaktiivse interpretaatori käsurealt avaldis, mille väärtuseks on lõpmatu list, mille elemendid on järjest kõik ainult A-tähtedest koosnevad lõplikud stringid alustades tühjaga.
172. Arvutada interaktiivse interpretaatori käsurealt kirjutatud ühe avaldise abil esimene arvu 18 aste, mis jagub 243-ga.
173. Arvutada interaktiivse interpretaatori käsureale kirjutatud ühe avaldise abil, mitme naturaalarvu faktoriaal on ülimalt 100-kohaline.
174. Realiseerida kõrgemat järku funktsioonide abil definitsiooniga (29) või (30) antav funktsioon kõikvõrdsed.

Lõpuks vaatleme ka üht näidet kõrgemat järku funktsiooni defineerimisest.

Koodiga (40) oli defineeritud arvude astendamine täisarvuga logaritmilise keerukusega. Astendamise kui ühe fikseeritud objektiga sooritatud sama operatsiooni paljukordse sooritamise näiteid on matemaatikas palju, mistõttu on mõttekas see üldstruktuur välja abstraherida. Kirjutame funktsiooni `kõrgAste`, mis suudab leida mistahes etteantud assotsiatiivse binaarse operatsiooni naturaalarv korda sooritamise tulemuse. Lisaks astendamise kahele argumendile võtab ta argumendiks ka binaarse operatsiooni, millega kokkuarvutamine toimub, ja tema ühikelemendi, mis tuleb vastuseks anda 0-ga astendamise puhul. Jätame vaatluse alt välja astendamise negatiivse täisarvuga, mille jaoks meil oleks vaja veel ka pöördoperatsiooni.

Otse koodi (40) järgi tehes saame definitsiooniks

```
kõrgAste (*) e a n
= case compare n 0 of
  GT
    -> let
      (q , r) = divMod n 2
      z = kõrgAste (*) e a q
    in
      if r == 0 then z * z else z * z * a
  EQ
    -> e
  _
    -> error "kõrgAste: negatiivne astendaja"
(62)
```

kus `kõrgAste` esimene argument `*` on operatsioon ja `e` arvuga 0 astendamise tulemus. Argumendid `a` ja `n` on, nagu ka definitsioonis (40), astme alus ja astendaja, st objekti `a` arvutatakse `n` korda operatsiooniga `*` kokku. Pangem tähele, et sümbol `*` tähendab definitsioonis (62) kõikjal seda operatsiooni, mis tuleb `kõrgAste` argumentiks, mitte tavalist korrutamist, sest ta seotakse `kõrgAste` argumentinäidises.

Tavalise astendamise saab nüüd defineerida `kõrgAste` kaudu, fikseerides operatsiooniks korrutamise ja 0-ga astendamise tulemuseks korrutamise ühiku 1:

```
aste = kõrgAste (*) 1.
(63)
```

Definitsioonis (63) tähendab `*` tavalist korrutamist, sest ta pole lokaalselt seotud muus tähenduses. Soovi korral võime anda `kõrgAste` argumentideks muid väärtusi, defineerides näiteks korrutamise korduva liitmisoperatsiooni tulemusena alates 0-st:

```
korrutis = kõrgAste (+) 0.
```

Muutuja `korrutis` väärtustamisel antakse funktsiooni `kõrgAste` argumentiks `*` liitmine.

Koodi (62) korrektseks kasutamiseks peab `e` väärtuseks andma operatsiooni vasakpoolse ühiku. Vastasel korral hakkab see tulemust mõjutama. Kui operatsioonil on ainult parempoolne ühik, tuleb selle kasutamiseks panna koodis `z * z * a` asemele `a * z * z`.

Veelgi olulisem 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 (62) arvutamine korrektseid tulemusi vaid assotsiatiivsete operatsioonide korral.

Näiteks ei ole võimalik koodiga (62) defineeritud funktsiooni `kõrgAste` rakendusena saada superastendamist, mis seisneb tavalise astendamise korduvas sooritamises, sest astendamine ei ole assotsiatiivne. Pealegi pole astendamisel vasakpoolset ühikelementi (parempoolne siiski on, 1, nii et sellest probleemist saaks üle).

Kõrgemat järku astendamise abil saab arvutada asju, mis esmapilgul ei tundu sellega kuidagi seotud olevat. Vaatame näiteks operatsiooni \otimes täisarvupaaridel, mis defineeritakse seosega

$$(a, b) \otimes (c, d) = (ad + bc - ac, ac + bd). \quad (64)$$

Märkame, et

$$(F_n, F_{n+1}) \otimes (1, 1) = (F_n + F_{n+1} - F_n, F_n + F_{n+1}) = (F_{n+1}, F_{n+2}),$$

st suvalist paari kahest järjestikusest Fibonacci arvust paariga $(1, 1)$ opereerides saame järgmise paari kahest järjestikusest Fibonacci arvust. See näitab, et kõik paarid kahest järjestikusest Fibonacci arvust on saadavad algpaarist $(0, 1)$ järjest paariga $(1, 1)$ opereerides. Et $(0, 1)$ on operatsiooni \otimes ühikelement, siis on tegemist paari $(1, 1)$ astmetega.

Osutub, et operatsioon \otimes on assotsiatiivne. Seetõttu võib temale vastavate astmete arvutamiseks kasutada koodiga (62) defineeritud funktsiooni `kõrgAste`, mis annab meile viisi Fibonacci arvude arvutamiseks logaritmilise keerukusega:

```
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) \quad (65)
```

Definitsioon (65) ületab oma efektiivsusest kõik varasemad, ka need, mis lasid Fibonacci arve listist otsida, sest listist otsimine on lineaarse keerukusega.

Tuntud on matemaatikas ka funktsioonide astendamine, kus korratavaks operatsiooniks on kompositsioon. Haskellis on kompositsioon defineeritud infiksoperaatori `.` väärtuseks. Kompositsiooni ühik on samasusfunktsioon, mis on prelüüdis defineeritud muutuja `id` väärtuseks. Kuna kompositsioon on assotsiatiivne, väljendub funktsioonide astendamine seega avaldisega `kõrgAste (.) id`.

Kuid funktsioonide astendamise juures ei anna astendamiseks “jaga ja valitse” tehnika kasutamine efektiivsusevõitu. 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 $g \ . \ g$, siis erinevalt tavalise astendamise juhust pole siin mingit kasu asjaolust, et operatsiooni argumendid on võrdsed, sest rakendades seda kompositsiooni argumendile, sooritatakse kõigepealt üks g sellel argumendil ja teine g saadud tulemusel, mitte samal argumendil, mis võimaldaks arvutusressurssi kokku hoida. Kokkuvõttes sooritatakse funktsiooni f astme rakendamisel argumendile ikka astendajaga võrdne arv f rakendamisi.

Ülesandeid

175. Tõestada, et valemiga (64) defineeritud operatsioon \otimes on assotsiatiivne ja $(0, 1)$ on tema kahepoolne ühik.
176. Kirjutada oma moodulisse definitsioon (62) ja testida funktsiooni `kõrgAste` interaktiivse interpretaatori käsurealt erinevate operatsiooniargumentidega.
177. Tuginedes funktsioonile `kõrgAste`, kirjutada interaktiivse interpretaatori käsurealt avaldis, mille väärtus saadakse siinuse 100-kordsel rakendamisel arvule 1.
178. Defineerida koodiga (39) antud lineaarse astendamise eeskujul kõrgemat järku funktsioon `kõrgLinAste`, mis oleks sama tüüpi nagu `kõrgAste` ja millega saaks arvutada ka superastendamist.
179. Defineerida kõrgemat järku funktsioon `kõrgFact`, mis võtab järjest argumendiks operatsiooni, tema parempoolse ühiku ja naturaalarvu ning arvutab arvud $n, n-1, \dots, 1$ selle operatsiooniga kokku. Näiteks `kõrgFact (^) 1 n` väärtuseks peab olema $n^{(n-1)\dots 1}$. Defineerida operatsiooni `kõrgFact` kaudu faktoriaalifunktsioon ning binoomkordajaid $\binom{n}{2}$ leidev funktsioon.
180. Defineerida funktsioon `filterNext`, 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.
181. Defineerida funktsioon `countFilter`, mis võtab järjest argumendiks predikaadi p ja listi l ning annab välja paari, mille esimene komponent võrdub avaldise `filter p l` väärtusega ja teine komponent näitab eemaldatud elementide arvu algse listiga võrreldes. Definitsioon peab `countFilter p` esitama kujul `foldr ⊕ e`.
182. Defineerida funktsioon `countDropWhile`, mis võtab järjest argumendiks predikaadi p ja listi l ning annab välja paari, mille esimene komponent võrdub avaldise `dropWhile p l` väärtusega ja teine komponent näitab ära visatud elementide arvu algse listiga võrreldes. Selle definitsiooni järgi arvutamine peab listi läbima ülimalt üks kord.
183. Defineerida funktsiooni `foldl` teisend `foldl'`, mis väärtustab akumulaatorit jooksvalt.
184. Defineerida standardfunktsioon `scanl` rekursiivse listiga. Testida mittekommutatiivsete operatsioonidega.
185. Defineerida funktsioon `minuZipWith`, mis töötab nagu `zipWith`, aga ei viska ülejäävat listisaba ära, vaid jätab selle tulemuslisti lõppu.
186. Defineerida kõrgemat järku funktsioon `mapPair`, mis võtab järjest argumendiks *curried*-kujul funktsiooni \oplus ja listi l ning mille tulemuseks on list, mille ta saab, võttes listi l elemendid kahekaupa paaridesse ja rakendades igale paarile operatsiooni \oplus .

187. Defineerida kõrgemat järku funktsioon `foldt`, mis võtab argumendiks operatsiooni \oplus ja listi l ning arvutab listi l elemendid operatsiooniga \oplus kokku, asetades sulud selliselt, et avaldise struktuur oleks 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 $a_1, a_2, a_3, a_4, a_5, a_6, a_7$, on tulemuseks $((a_1 \oplus a_2) \oplus (a_3 \oplus a_4)) \oplus ((a_5 \oplus a_6) \oplus a_7)$.
188. Realiseerida sorteerimine mestimismeetodil ülesannetes 186 ja 187 defineeritud funktsioonide `mapPair` ja `foldt` abil.
189. Defineerida kõrgemat järku sorteerimine, mis võtab võrdlusoperatsiooni argumendiks, kiirmeetodil ja mestimismeetodil.
190. Defineerida funktsioon `rotatsioonid`, mis leiab argumentlisti kõik tsüklilised nihked.