Haskelli interaktiivsete interpretaatorite kasutamine

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

Meil on olemas interaktiivsed keskkonnad Hugs ja GHCi, milles saab Haskell-programme otse testida. Programmitekst tuleb muidugi enne faili kirjutada ja interaktiivses keskkonnas see fail sisse lugeda. Mainitud interaktiivsete keskkondadega tulekski esimeses järjekorras tutvust teha.

Hugs

Hugs on Haskelli interpretaator, tema koduleht asub aadressil

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

- Käivitada Hugs, andes käsurealt käsu `hugs`.
- Teises aknas visata silm peale Hugsi standardteekidele, mille leiab ülikooli arvutivõrgu failisüsteemis kataloogist

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

- Teha endale selgeks Hugsi põhikäsud

  `:q` väljumine,
  `:e` tekstiredaktori avamine,
  `:l` mooduli sisselugemine (argumendiks failinimi),
  `:r` viimase sisselugemiskäsu kordamine,
  `:?` Hugsi 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. Hugsi `:e` on hea selle poolest, et kui editorist väljuda, sooritab Hugs automaatselt ka kompileerimise.

- Hugsi vaikimisiparameetreid saab sättida ka keskkonnamuutujaga `HUGSFLAGS`. Näiteks minul on kodukataloogi failis `.login` rida

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

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

- Lasta Hugsil väärtustada lihtsaid Haskell-avaldisi.
  - Mõni aritmeetiline avaldis, nt `2 + (-3)` vms.
  - Mõni n-ö "funktsioone sisaldav" avaldis, nt `6 * asin 0.5`, see peaks andma π. Kuna π on ka standardteegis muutuja `pi` väärtuseks, saate seda kontrollida. Näiteks võrrelge neid avaldisi võrdusoperaatoriga == või käskige ühest teine lahutada.
  - Mooduli Data.Char (kuna see moodul on Haskell-98 standardis, saab ta kätte ka ilma paketinimeta Data) eksporditavad funktsioonid `ord` ja `chr` konverteerivad sümbolite ja nende koodide vahel. Võiks proovida väärtustada avaldisi nagu `ord 'A'`, `chr 7` jms. Mis juhtub, kui moodul pole sisse loetud?

GHCi

GHC on Haskelli kompilaator, mille koduleht asub aadressil

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

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

- Käivitada süsteem käsurealt käsuga `ghci`.
- Standardteegid on vaid kompileeritud kujul, algtekste näha ei saa. Standardteegid on sarnased Hugsi omadega, kuid leidub ka erinevusi (meie kursuses enamik neid tõenäoliselt ei tähelda).
- Interaktiivse keskkonna käsud `:q`, `:?`, `:s` on sarnased Hugsiga. Standardteekide sisselugemisel tuleb kasutada käsku `:m`, oma failide jaoks `:l` ja `:r` nagu Hugsis.
- Proovida ka GHCi-s väärtustada lihtsamaid Haskell-avaldisi.

Haskelli süntaksi põhielemendid

Haskell-programmi osad klassifitseeruvad, peale võtmesõnade ja -sümbolite, peamiselt **avaldisteks**, **näidisteks** ja **deklaratsioonideks**.

Avaldistel on (kõigi identifikaatorite defineerituse korral) olemas **väärtus**. Näidis võimaldab etteantud väärtuse järgi määrata näidise komponentide väärtusi. Deklaratsiooni ülesanne on midagi uut defineerida.

Näidis saab komponentidena sisaldada ainult näidiseid. Avaldis võib sisaldada nii näidiseid, avaldisi kui deklaratsioone. Uute identifikaatorite väärtusi defineeriv deklaratsioon sisaldab tavaliselt näidiseid (**vasakus pooles**) ja avaldisi (**paremas pooles**), kuid võib sisaldada ka deklaratsioone.

Programmitekst on deklaratsioonide kogum.

Pange tähele, et funktsionaalses paradigmas ei saa muutujate väärtusi muuta. Muutujad siin ei ole muutujad sellises mõttes, nagu nad on imperatiivses programmeerimises, pigem sellises mõttes, nagu nad on matemaatikas. Muutujad võivad võtta teised väärtused, kui sama blokki uuesti läbitakse, kuid mingit omistamisoperatsiooni, millega saaks bloki sees töö käigus väärtust muuta, ei ole ja selline asi on põhimõtteliselt võimatu.

Andmemuutujate defineerimine

Vaatame siin kõige põhilisemaid deklaratsioone, mis defineerivad andmemuutujaid. Peale nende on võimalik defineerida ka tüübimuutujaid, uusi tüüpe ja väärtusi ning tüübiklasse.

Kõik selle osa deklaratsioonid võiks koondada faili nimega `Syntax.hs`.

- Lihtsaimal juhul on näidis lihtsalt muutuja:

```
a = 10
```

  - Kirjutada see rida programmi. Kuidas interaktiivses keskkonnas seda testida?
  - Lisada deklaratsioon, mis defineerib muutuja `e` väärtuseks e, kasutades standardset eksponentfunktsiooni `exp`.

- Keerulisemal juhul on näidis konstrueeritud mitmest osast, mis on omakorda näidised:

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

  See deklaratsioon määrab `x` väärtuseks 5 ja `y` väärtuseks 0.2.

  - Kirjutada analoogiline deklaratsioon, mille vasakus pooles on keerulisem näidis.
  - Standardfunktsioon `properFraction` võtab argumendiks ujukomaarvu ja annab väärtuseks paari tema täis- ja murdosaga. Defineerida muutujad `q` ja `r`, mis on vastavalt arvu e^{10} täis- ja murdosa. Mis juhtub, kui astendaja on suurem, nt 100? Arvutada analoogselt $e^{e^{e}}$ täis- ja murdosa.

- Funktsioonide defineerimiseks siiani õpitud meetodid ei kõlba. Kui on vaja deklaratsiooni, mis defineeriks funktsiooni, on üheks võimaluseks koostada vasak pool mitmest näidisest: esimene, mis peab sel juhul kindlasti olema muutuja, tähistab defineeritavat funktsiooni ja järgnevad, tühikuga ja vajadusel ka sulgudega eraldatult, tema argumente. Näiteks

```
sqr x = x * x
plus (x , y) = x + y
```


  - Kirjutada funktsioonid `vasak` ja `parem`, mis võtavad argumendiks paari ja annavad tulemuseks vastavalt vasaku ja parema komponendi.

  Kui mõne muutuja väärtust arvutamisel ei vajata, võib selle muutuja vasakus pooles asendada alakriipsuga. Alakriipsu nimetatakse **jokkeriks** ja see on lisaks seniõpitutele veel üks võimalik näidis.

  - Kirjutada funktsioonide `vasak` ja `parem` definitsioonid nii ümber, et ühtki muutujat ei reserveeritaks mittekasutatavate väärtuste jaoks.
  - Funktsioonid `vasak` ja `parem` on aga standardteegis tegelikult olemas — vastavalt `fst` ja `snd`. Kuidas seda kontrollida?
  - Kirjutada funktsioon `imelik`, mis võtab argumendiks paari ja annab välja paari, mille esimene komponent on argumentpaari esimene komponent ja teine komponent on argumentpaar ise. Mis jääb kripeldama?

  Kui paremas pooles vajatakse lisaks muutujate väärtustele ka mõne suurema vasakus pooles esineva näidise väärtust, võib ka selle jaoks vasakus pooles muutuja sisse tuua, kirjutades selle muutuja lihtsalt vajaliku näidise ette ja eraldades selle muutuja näidisest sümboliga `@`, mille tähendus on niisiis defineeriv võrdus. Niimoodi moodustatud näidist nimetatakse **kuinäidiseks**, mis on veel üks võimalik näidiseliik.

  - Kirjutada kuinäidise abil eelmise punkti funktsioon ümber. Kasutada jokkerit igal pool, kus võimalik.

- Funktsioone võib defineerida aga ka nii, et vasakus pooles on vaid üks näidis. Muidugi peab see olema muutuja, sest funktsioon ei esitu struktuurina mingitest komponentidest (vähemalt Haskellis). Parem pool aga peab olema avaldis, mille väärtus on funktsioon. Selleks on olemas nn **lambdaavaldised**, mis sisaldavad iseendas nii funktsiooni definitsiooni vasaku kui parema poole. Näiteks ülaltoodud funktsioonid `sqr` ja `plus` saanuks lambdaavaldistega defineerida nii:

```
sqr = \ x -> x * x
plus = \ (x , y) -> x + y
```

  Üldskeem peaks siit selge olema. Anda harjutuseks ka funktsioonidele `vasak` ja `imelik` selline definitsioon, jätkates ka jokkeri ja kuinäidise kasutamist. Eelmisi funktsioone mitte puutuda, vaid anda uued nimed, nt `vasak'` ja `imelik'`.

- Avaldistes, mis muidu läheksid liiga pikaks, saab osade väärtusi omistada lokaalsetele muutujatele, kasutades ***let*-avaldist**. Lokaalsete muutujate skoop hõlmab parajasti kogu selle *let*-

avaldise.

```
letnäide x y
   = let
        a = x * x + 3 * x * y + 2 * y * y
        b = sin x - 3 * sin x * cos y + cos x
     in
     b / a * 100
```

Samuti on kasulik kasutada *let*-avaldist, kui mõni arvutusi sisaldav avaldis esineks muidu korduvalt: see võimaldab korduvatest arvutustest vabaneda.

– Kirjutada funktsioon `summanali`, mis arvutab oma argumentide summa siinuse ja summa enda suhte. Sama summat ei tohi arvutada korduvalt. Kõik vajalikud oma abimuutujad defineerida lokaalselt.

- Tõenäoliselt olete proovinud juba kirjutada **tingimusavaldisi**. *If*-avaldis on lihtne ja tema kirjutamine ei nõua erilist õpetamist. Tuleb vaid tähele panna, et Haskellis peab *if*-avaldisel alati olema nii *then*- kui *else*-haru ning need peavad olema sama tüüpi. Samuti võib probleeme tekkida halva paigutusega, mille tagajärjel kompilaator ei saa *if*-avaldisest aru.

  – Kirjutada funktsioon `verbsgn`, mis arvulise argumendi korral annab väärtuseks `"Positiivne"`, `"Negatiivne"` ja `"Null"` vastavalt argumendi väärtusele.

- Tihti, eriti kui tingimusi on rohkem kui üks, on tingimusavaldiste asemel mugavam kasutada **valvureid**. Iga valvur esitab püstkriipsu järel ühe tingimuse ning seejärel võrdusmärgiga eraldatult avaldise, mis tuleb valida selle tingimuse täidetuse korral. Valvurid oleks sobilik rajastada üksteise alla.

  – Anda funktsioonile `verbsgn` valvuritega definitsioon, mis *if*-avaldist ei sisalda.

- Kui mõnd avaldist kasutatakse korduvalt mitmes valvuris või neile järgnevates avaldistes, siis *let*-avaldist pole võimalik kasutada, kuna parem pool ei moodusta kokku avaldist. Selleks puhuks on olemas ***where*-konstruktsioon**, mis võimaldab defineerida üle kogu parema poole (mis hõlmab kõiki valvureid ja nendele vastavaid avaldisi) nähtavaid lokaalseid muutujaid. *Where*-konstruktsioon kirjutatakse kogu parema poole lõppu (st üldjuhul alla) võtmesõna **`where`** järele.

  – 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, siis anda seda ütlev tekst. Ühtki avaldist ei tohi arvutada korduvalt. Kõik vajaminevad oma abimuutujad defineerida lokaalselt. Standardfunktsioonidest läheb vaja funktsioone `isUpper`, `isLower` ja `isDigit`.

- Kõige üldisem kontrolle sisaldav konstruktsioon on ***case*-avaldis**, mis võimaldab enda sees kirjutada kõiki siiani nähtud tingimuskonstruktsioone. Väline otsustus tehakse näidiste sobitamisega. Näiteks järgmine triviaalne funktsioon teisendab tõeväärtuse vastavaks arvuks.

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

Süntaksi poolest on *case*-avaldise igale näidisele vastav juht nagu tavalise definitsiooni parem pool, milles võrdusmärkide asemel on nooled.

Toodud näiteavaldise puhul käib täitmine järgnevalt. Sisse tuleb tõeväärtusavaldis. Kontrollitakse, kas see sobitub järjekorras esimese näidisega, milleks on `True`, ehk teisi sõnu, kontrollitakse tingimuse täidetust. Kui sobitumine õnnestus, st tingimus oli täidetud, siis loetakse funktsiooni väärtuseks sellele näidisele vastava parema poole väärtus ning ülejäänud variante ei uurita. Kui sobitumine ei õnnestunud, võetakse järjekorras järgmine näidis jne. Antud juhul on järgmiseks näidiseks jokker, millega sobitub iga avaldis. Seega suvalisel juhul, millel üldse selle sobitamiseni jõutakse (st tingimus ei olnud täidetud), minnakse edasi sellele näidisele vastavasse paremasse poolde.

– Kirjutada täisarve tõeväärtuseks teisendav funktsioon `intValue`, mis annab veateate, kui argument pole 0 ega 1. Haskelli funktsioon `error`, mis võtab argumendiks stringi, lõpetab programmi töö, andes ekraanile veateatena selle stringi.

– Anda funktsioonile `verbsgn` uus definitsioon (kasutada uut nime, nt `verbsgn'`), mis valvurite asemel kasutab *case*-avaldist. Kasutada standardset `Ordering`-tüüpi, mille väärtused on `GT`, `EQ` ja `LT`, ja standardfunktsiooni `compare`, mis kahe suuruse võrdlemise järgi annab ühe neist kolmest väärtusest.

– Kirjutada funktsioon `kolmkoos`, mis võtab argumendiks täisarvu. Kui argument on positiivne, siis käitub nagu `sümbolid`. Kui argument on negatiivne, siis käitub nagu `verbsgn`. Kui argument on null, siis lõpetab veateatega, et argument oli null. Esimeses lähenduses võiks *case*-avaldisest kutsuda välja viidatud funktsioone ja testida *case*-avaldise korrektsust, seejärel aga harjutamiseks, kuidas suuremaid konstruktsioone üksteise sisse panna, loobuda sellest ja kopeerida nende funktsioonide kehad otse *case*-avaldisse.

- Üht ja sama funktsiooni saab Haskellis defineerida ka mitme deklaratsiooniga. Sellise "multideklaratsiooni" puhul proovib süsteem väärtustamisel deklaratsioone järjest ülalt alla ja valib parema poole sellel deklaratsioonil, millel esimesena kõik näidised sobituvad. Kui niisugust deklaratsiooni ei leita, tekib täitmisaegne viga.

  Niisugune definitsioon tuleb kõne alla juhul, kui muidu koosneks funktsiooni keha ühest *case*-avaldisest nagu ülaltoodud näitefunktsiooni `boolValue` puhul ning hargnemine toimub argumendi väärtuse järgi. Näiteks alternatiivselt võinuks selle funktsiooni defineerida järgnevalt.

```
boolValue' True
   = 1
boolValue' _
   = 0
```

Tuleb tähele panna, et sama funktsiooni defineerimisel mitme deklaratsiooniga peab ilmutatult toodud argumentide arv kõigis neis deklaratsioonides sama olema, vastasel korral loetakse see staatiliseks veaks, programm ei kompileeru.

– Kirjutada ülalpool spetsifitseeritud funktsiooni `intValue` definitsioon, milles *case*-avaldise asemel on kasutatud mitut deklaratsiooni.

- Enamasti suudab Haskell avaldiste tüübid ise tuletada ja neid pole programmeerijal vaja kirjutada. Mõnikord osutub vajalikuks täpsustada avaldise tüüpi ilmutatult. Kirjutage näiteks deklaratsioon

```
f x = show (x :: Int).
```

Standardfunktsioon `show` teisendab oma argumendi stringiks. (Muide, vaikimisihäälestuse korral kasutab Hugs seda funktsiooni avaldiste väärtuste esitamiseks kasutajale.) Niisiis `f` esitab oma argumendi stringina, kuid käsitledes teda kuuluvana tüüpi `Int`, mis on 32-bitiste märgiga täisarvude tüüp. Kontrolliks võiks lasta väärtustada avaldisi `f` x, kus x on mõni väike arv, ja seejärel, kus näiteks x = `4294967295`. Võrdluseks laske väärtustada ka `show 4294967295` — kui ilmutatud tüübikitsendusi pole antud, käsitleb Haskell arvulisi literaale vaikimisi kuuluvana tüüpi `Integer`, mis on kuitahes pikkade täisarvude tüüp.

Kuna Hugs kasutab funktsiooni `show` suvalisele väärtustamiskäsule vastamisel, ei olnud selle funktsiooni ilmutatud lisamisel suuremat mõtet. See oli lihtsalt *show*. Oleks võinud lasta väärtustada `4294967295 :: Int` ja `4294967295`.

- Tüüpe võib eraldi deklaratsioonis näidata ka defineeritavatele muutujatele. Sellist deklaratsiooni nimetatakse **tüübisignatuuriks**. Kuigi tüübisignatuure Haskelli poolest enamasti vaja anda ei ole, aitavad need tugevalt kaasa programmi tekstist arusaamisele, mistõttu soovitav on need lõpuks ikkagi lisada.

– Anda tüübisignatuurid mõnedele eelnevalt defineeritud muutujatele, nagu näiteks `verbsgn` ja `sümbolid`. Kontrollige, kas programm kompileerub.

Vajaliku tüübisignatuuri saate defineeritud muutujatele Hugsis ja GHCi-s käsuga `:i` (argumendiks muutuja). Ise võib anda muutujatele sama või laiema tüübi. Aga tüüpidega tegeleme lähemalt kunagi hiljem.

- Haskellis programmeerimisel on asendamatuteks andmestruktuurideks listid. **List** on lõplik või lõpmatu ahel sama tüüpi elementidest, programmeerimise terminoloogias üks võimalik dünaamiline andmestruktuur.

Listidega seonduvad ennekõike konstruktorid `[]` ja `:`, mis tähendavad vastavalt tühja listi ja listi algusse elemendi lisamise operatsiooni. Kui list esitub kujul $x : l$, siis öeldakse, et x on selle listi pea ja l selle listi saba.

Näiteks x `:` `[]` märgib listi, mille ainus element on x, ja x `:` y `:` z `:` `[]` märgib 3-elemendilist listi, mille elemendid pea poolt saba poole on x, y ja z. Viimasest näitest nähtub, et operaator `:` on **paremassotsiatiivne**, st sulgudeta kirjutis a `:` b `:` c tähendab sama mis a `:` $(b$ `:` $c)$.

Kuna `:` on konstruktor, siis on meil võimalik kasutada näidiseid kujul $x : l$. Triviaalne näide oleks deklaratsioon

```
x : xs = 1 : 3 : [],
```

mis annab muutujale `x` väärtuseks 1 ja muutujale `xs` väärtuseks 3 `:` `[]`, st listi, mille ainus element on 3. Pisut sisukam on järgmine:

```
head (x : _)
   = x
head _
   = error "head: empty argument list"
```

Need deklaratsioonid defineerivad funktsiooni `head`, mis leiab mittetühja argumentlisti korral tema pea, tühja argumentlisti korral aga lõpetab veateatega. See funktsioon on juba moodulis Prelude olemas. Samuti on seal näiteks järgmine lihtne definitsioon (või midagi samaväärset).

```
null []
   = True
null _
   = False
```

– Testida interaktiivse interpretaatori käsurealt standardfunktsioone `head` ja `null`. Mida teeb funktsioon `null`?

– Kirjutada funktsioon `esipaar`, mis võtab argumendiks listi ning, kui selles on vähemalt 2 elementi, tagastab esimesed kaks elementi paarina, vastasel korral lõpetab veateatega.

– Kirjutada funktsioon `primLoenda`, mis võtab argumendiks listi ning annab väärtuseks `"Null"`, kui list oli tühi, `"Üks"`, kui listis on täpselt üks element, ja `"Palju"`, kui listis on rohkem elemente.

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 `:` ja `[]`.

Põhilisim neist on listi esitamine komadega eraldatud elementide loeteluna kantsulgudes. Näiteks ülalvaadeldud listid x `: []` ja x `:` y `:` z `: []` on selles notatsioonis vastavalt `[`x`]` ja `[`x`,` y`,` z`]`. Ülaltoodud näitedeklaratsioon oleks aga samaväärselt

```
x : xs = [1, 3]
```

Seda süntaksit lubab Haskell kasutada ka näidistes. Järgnev näide defineerib funktsiooni `singleton`, mis otsustab, kas argumentlist on üheelemendiline.

```
singleton [_]
  = True
singleton _
  = False
```

– Anda funktsioonile `primLoenda` samaväärne definitsioon, mis ei sisalda konstruktorit `:`.

- Stringid on Haskellis sümbolite listid. Seda võib arvestada, kui stringe kasutavaid funktsioone defineeritakse. Juba tuttav kirjapilt katkematu sümbolite joruga jutumärkides on stringide erisüntaks, mis on kasutusel ainult mugavusest. 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.

– Kirjutada funktsioon `esiSuureks`, mis teisendab oma argumentstringi esimese tähe suureks. Abiks on standardfunktsioon `toUpper` moodulist $\mathrm{Data.Char}$.

Ka stringide erisüntaksit saab kasutada näidistes. Praktiliselt tuleb seda siiski harva ette, kuna niisugusel puhul pole võimalik näidata varieeruvaid komponente, vaid ainult konstantseid stringe.

– Püüda täpselt aru saada, mida viimase lausega on mõeldud. Kui kirjeldatud põhjus jääb arusaamatuks, teha see omale n-ö omal nahal selgeks, üritades kirjutada funktsioonile `primLoenda` alternatiivset versiooni, mis võtaks argumendiks vaid stringe, ja kasutades näidistena vaid stringinäidiseid.

Mõned eritehnikad

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

- **Rekursioon**.

  Kuna muutujate väärtuse muutmine on võimatu, ei saa olemas olla ka tavapäraseid tsükleid. Kõik tsüklilised protsessid tuleb programmeerida rekursiooniga. **Rekursiivseks** nimetatakse definitsiooni, mille parem pool sisaldab muutujat, mida see definitsioon defineerib.

  Kirjutades rekursiivset funktsiooni, tuleb funktsiooni käitumine suuremate argumentide korral avaldada funktsiooni väärtuste kaudu väiksematel argumentidel. Lõpuks on vaja spetsifitseerida ka baasjuht, milles rekursiivset pöördumist ei toimu ja millest arvutus saab seega "peale hakata". Vajadusel tuleb lisada veel erijuhte, nagu näiteks veateade negatiivsetel argumentidel.

  – Kirjutada funktsioon `fact`, mis leiab oma argumendi faktoriaali; negatiivse argumendi puhul tuleb lõpetada olukorda identifitseeriva veateatega.

    Kasutades eelnevalt õpitud süntaktilisi konstruktsioone, püüda kirjutada definitsioon nii, et väärtustamine toimuks võimalikult efektiivselt. Arvestada, et iga näidiste sobitamine ja iga tingimuse kontroll võtavad aega.

  – Kirjutada analoogilised funktsioonid `subFact` ja `supFact`, mis argumendil n annavad väärtuseks vastavalt arvud $1 + 2 + \ldots + n$ ja $n^{(n-1)^{\cdots^{1}}}$.

  Kuna listid on oma olemuselt rekursiivsed elukad (mittetühja listi saba on omakorda list), siis peaaegu kõik sisukad funktsioonid listidel tuleb programmeerida rekursiooniga.

  – Kirjutada funktsioon `vahed`, mis leiab etteantud listi järgi listi, mis koosneb argumentlisti kahe järjestikuse elemendi vahedest.

  Mitu funktsiooni võivad olla üksteisega **vastastikku rekursiivsed**. See tähendab, et isegi kui funktsiooni kehas ilmutatult teda välja ei kutsuta, võib töö käigus niisugune pöördumine siiski mõne teise funktsiooni kaudu aset leida. Näiteks järgmised funktsioonid `üksnul` ja `nulüks` on omavahel vastastikku rekursiivsed, sest nad kutsuvad kumbki teineteist välja.

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

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

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

- Oletame jätkuvalt, et meile pakub huvi ainult funktsioon `üksnul`. Siis võiks otsida talle definitsioon, mis funktsiooni `nulüks` ja vastastikust rekursiooni ei vaja. Kas oskate sellise kirjutada?
- Kirjutada funktsioon `vahelduv`, 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.

- **Kõrgemat järku funktsioonid**.

  Funktsionaalses keeles võib iga funktsioon olla teise funktsiooni väärtuseks ja ka argumendiks. See võimaldab sarnase skeemi järgi defineeritud funktsioonidest skeemi välja abstraheerida ja seega koodi kordamist vältida, andes vajalikud parameetrid lisaargumentidena sisse.

  Haskellis on n-ö kõrgemat järku funktsioone väga lihtne kasutada ja defineerida. Kindlasti olete, ise seda teadmata, neid juba kasutanud või isegi defineerinud. Näiteks standardfunktsioon `compare` kasutus (`compare` x y) näitab, et tema väärtuseks argumendil x on funktsioon, mis omakorda võttes argumendiks y annab tulemuseks x ja y võrdluse tulemuse. Samasugused on kõik infiksoperaatorid. Näiteks liitmisoperatsioon + võtab argumendiks mitte paari, vaid ühe arvu a ja annab tulemuseks funktsiooni, mis omakorda võtab argumendiks b ja annab tulemuseks $a + b$. Infikssel kasutamisel (operatsioon oma argumentide vahel) ei paista see välja, aga prefikssel kasutamisel (funktsioon oma argumendi ees; operaatorimärgile tuleb siis sulud ümber panna, näiteks `2 + 4` prefiksselt samaväärselt ümber kirjutatuna on `(+) 2 4`) tuleb sellega arvestada.

  - Anda interaktiivse interpretaatori käsurealt avaldisi, milles infiksoperaatoreid (`+`, `*`, – jms) kasutatakse prefiksselt.

  Ka täheliste nimedega funktsioone, mis võtavad piisavalt argumente, on võimalik kasutada infiksselt. Infikssel kasutamisel tuleb funktsiooninimi panna tagurpidi ülakomade vahele; näiteks `compare 2 4` infiksselt samaväärselt on ``2 `compare` 4``.

  - Anda interaktiivse interpretaatori käsurealt avaldisi, milles täheline funktsiooninimi on kasutatud infiksselt. Näiteks funktsioone `div` ja `mod`, mis leiavad kahe täisarvu täisarvulist jagatist ja jääki, kasutatakse tihti infiksselt. Püüda neid kasutada nii prefiksselt kui infiksselt.

  - Eelmises punktis kirjutatud funktsioonid `fact`, `subFact` ja `supFact` on defineeritud ühe ja sama skeemi kohaselt. Erinevad vaid operaator, millega arve kokku opereeritakse, ja algväärtus, millest arvutamine pihta hakkab.

    Kirjutada kõrgemat järku funktsioon `kõrgFact`, mis, sõltuvalt esimestest argumentidest, milleks on operatsioon ja algväärtus, arvutab kolmanda argumendina sisse tuleva arvu n korral arvud 1 kuni n selle operatsiooni abil kokku, kasutades opereerimise algväärtusena teist argumenti.

    Muuhulgas peavad eelpool defineeritud funktsioonid `fact`, `subFact` ja `subFact` olema saadavad erijuhtudena funktsioonist `kõrgFact`:

```
fact = kõrgFact (*) 1
subFact = kõrgFact (+) 0
supFact = kõrgFact (^) 1
```

    Veenduda konkreetsete avaldiste väärtustamise teel, et funktsioon tõesti teeb seda, mis nõutud.
  - Tõenäoliselt on teie kirjutatud definitsioonides funktsioonidel liigselt argumente, mille saab kavalalt manipuleerides ära koristada. Saage lahti võimalikult paljudest argumentidest definitsioonides, ilma funktsionaalsust muutmata.

  Funktsionaalses programmeerimises on kõrgemat järku funktsioonid üldlevinud nähtuseks.

  - Tutvuda loengumaterjali näidetega standardsetest listifunktsioonidest. Lisada võiks veel funktsiooni `elem`, mis argumentidel x ja l annab `True`, kui x esineb listis l, ja `False` vastasel korral. Järgnevas eeldame, et te neid tunnete.
  - Kirjutada funktsioonid
    * `takeLõpust`, mis töötab nagu `take`, aga võtab elemente listi lõpust;
    * `minuZipWith`, mis töötab nagu `zipWith`, aga ei viska ülejäävat listisaba ära, vaid jätab selle tulemuslisti lõppu;
    * `paarikaupaErinevad`, mis argumentlisti kohta otsustab, kas tema elemendid on paarikaupa erinevad;
    * `kõikVõrdsed`, mis argumentlisti kohta otsustab, kas tema elemendid on kõik võrdsed;
    * `disjunktsed`, mis kahe argumentlisti kohta otsustab, kas nad on disjunktsed, st tagastab `True`, kui listid ei sisalda ühiseid elemente, ja `False`, kui sisaldavad.

- **Akumulaatorid**.

  Akumulaatorid on rekursiivse funktsiooni argumendid, milles väärtustamise käigus asuvad arvutuste vahetulemused. Tihti on akumulaatoriteks spetsiaalselt selle eesmärgiga loodud lisaargumendid.

  See tähendab, et rekursiivsel pöördumisel antakse uuendatud vahetulemus selles lisaargumendis kaasa.

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

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

viib ruutkeerukusega arvutuseni.

Akumulaatori kasutamine võimaldab liste pöörata lineaarse keerukusega.

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

Siin lokaalse funktsiooni argument `as` on akumulaator.

Kui kirjutaksime Fibonacci arve leidva funktsiooni otse matemaatikast võetud rekurrentse valemi järgi Haskelli, näiteks kujul

```
fib 0
  = 0
fib 1
  = 1
fib n
  = fib (n - 1) + fib (n - 2)
```

toimuks arvutamine eksponentsiaalse keerukusega. Lineaarse keerukuse saavutamiseks tuleb pidevalt hoida kaks viimast vahetulemust korraga kättesaadavana. Üks võimalus selleks on defineerida abifunktsioon, mille väärtuseks oleksid paarid kahest järjestikusest Fibonacci

arvust.

```
fib
  = let
        fib 0
          = (0 , 1)
        fib n
          = let
                (u , v)
                  = fib (n - 1)
            in
            (v , u + v)
    in
    fst . fib
```

See variant pole ka kõige parem, sest igal rekursioonisammul konstrueeritakse uus paar, mis nõuab lisaressurssi. Teine võimalus on defineerida abifunktsioon, mis võtab kaks viimast vahetulemust akumulaatorargumentideks. See võimaldab paaride konstrueerimise arvelt kokku hoida. Praktiliselt annab see siiki vaid tühise ajavõidu.

- Defineerida `fib` lineaarse keerukusega, kasutades kahte akumulaatorit.
- Kas ka eelmises punktis kirjutatud listifunktsioone `paarikaupaErinevad` ja `disjunktsed` oleks kasulik kirjutada akumulaatoriga?

• Väärtustusjärjekorra ohjamine.

Faktoriaaliperre kuuluvate funktsioonide väärtuste arvutamine on teil tõenäoliselt programmeeritud nii, et ühtki vahetulemust ei saada kätte enne, kui rekursioon on jõudnud põhjani. See toob kaasa väärtustamata vahetulemuse ohjeldamatu pikenemise arvuti mälus, mille tagajärjena ei ole funktsioon kuigi suurte argumentide korral praktiliselt arvutatav. On selge, et tegelikult on võimalik vahetulemusi jooksvalt leida — iga inimene, kes käsitsi arvutab, teeb nii.

See viib mõttele kasutada `kõrgFact` mälukeerukuse optimeerimiseks akumulaatorit, milles hoida arvutuste vahetulemusi.

- Kirjutada funktsiooni `kõrgFact` lokaalne funktsioon ümber akumulaatoriga, milles asuvad arvutuste vahetulemused. (Mitte lõhkuda endist definitsiooni, vaid anda uuele muutujale uus nimi, nt `kõrgFact1`.)
- Võrrelda uue koodi tööd vanaga. Kas uus kood võimaldab funktsiooni väärtusi arvutada suurematel argumenidel kui vana? Milles on asi?

Väärtustamine saab olla kas laisk või agar. Funktsiooni argumendi **agar** väärtustamine tähendab, et argument väärtustatakse enne funktsiooni keha uurimist, **laisk** väärtustamine tähendab vastupidist, st minnakse kõigepealt funktsiooni kehha ja argument väärtustatakse ainult vajadusel. Haskellis tähendab ka väärtustamine, tõsi küll, tavaliselt mitte täielikku väärtustamist, nii et tavalise programmeerimiskeele mõttes oleksid ka Haskelli agarad funktsioonid laisad mis laisad.

Vaikimisi väärtustatakse Haskellis funktsiooni argument parajasti sellisel määral, et saaks kontrollida tema sobivust argumendinäidisega. Kui näidiseks on muutuja või jokker, sobib suvaline argument automaatselt, mistõttu funktsioon on sellise argumendi järgi ülaltoodud definitsiooni põhjal laisk.

Laiska argumenti saab sundida agaraks operaatoriga `$!`, mis käib funktsiooni ja argumendi vahele. Seega kui f on laisk funktsioon, siis $f\ \ x$ arvutamisel minnakse kohe f definitsiooni keha täitma, f `$!` x arvutamisel minnakse ka, kuid enne väärtustatakse x. See tähendab seda, et väärtustatakse x niikaua, kui kas selgub tema välimine konstruktor või tekib viga ning viimasel juhul lõpetab ka f `$!` x veaga.

Standardfunktsioon `const` on lihtsaim näide laisast funktsioonist. Ta võtab kaks argumenti ja annab välja neist esimese, visates teise argumendi ilma peale vaatamata minema. Teine argument võib seejuures olla avaldis, mis iseseisvalt väärtustades annaks vea või jääks lõpmatusse tsüklisse. Samas saab seda argumenti sundida ka väärtustama, kasutades tavalise funktsioonirakendamise asemel agarat rakendamist `$!`.

- Avaldise `log 0` väärtustamine lõpeb veateatega, sest nullist ei saa logaritmi võtta. Uurida, mis juhtub, kui talle rakendada `const 'A'` laisalt ja siis agaralt.
- Avaldis **`let`** `x = x + 1` **`in`** `x` jääb lõpmatusse tsüklisse, sest arv `x` on defineeritud rekursiivselt iseenda kaudu. Mis juhtub, kui sellele avaldisele rakendada `const 'A'` laisalt ja agaralt?

Niisugusteks katsetusteks on Haskelli prelüüdis defineeritud muutuja `undefined`, mis märgib suvalist tüüpi defineerimata väärtust (nn ***bottom***). Muidugi võib selle asemel kasutada avaldisi kujul `error` s, kus s on string, mis iseloomustab äratuntavalt seda koodikohta, kus ta asub.

- Modifitseerida funktsiooni `kõrgFact` akumulaatoriga definitsiooni nii, et akumulaatoris toimuks jooksev väärtustamine. Kui suurtel argumentidel nüüd funktsioon väärtuse välja suudab anda?
- Parandada samal põhimõttel ka funktsiooni `fib` definitsiooni.

• "**Jaga ja valitse**" tehnika.

- Kirjutada kõrgemat järku funktsioon `kõrgAste` astendamiselaadsete operatsioonide (korrutamine, astendamine, hüperastendamine, …; ühesõnaga, operatsioonid, kus üks argument ütleb, mitu korda tuleb teist argumenti iseendaga läbi "opereerida") sooritamiseks.

  Näiteks peab kehtima

  `kõrgAste (+) 0` $=$ `(*)`,

  `kõrgAste (*) 1` $=$ `(^)`.

- Mis on defineeritud funktsiooni keerukus?

"Jaga ja valitse" tehnika puhul jagatakse argument igal rekursioonisammul kaheks võimalikult võrdseks "osaks" ja kombineeritakse otsitav väärtus rekursiivse pöördumise tulemustest neil osadel. Mõnikord võimaldab see keerukuses võita.

Klassikaline näide "jaga ja valitse" tehnikast on nn kiirsorteerimine. Haskellis võiks see välja näha järgmiselt.

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

Siin `partition` on standardfunktsioon moodulist Data.List, mis jagab etteantud listi elemendid etteantud predikaadi järgi kaheks listiks: esimesse kogub need, mis predikaati rahuldavad, teise ülejäänud.

Proovides võib veenduda, et enamikul juhtudel tagab see definitsioon oluliselt kiirema töö kui sorteerimisalgoritmid, mis käsitlevad elemente ükshaaval.

- Eeldame, et `kõrgAste` kasutamisel antakse esimeseks argumendiks ainult assotsiatiivseid operatsioone. Siis sulgude paigutus pikas opereerimiste ahelas pole oluline ja opereerimisi võib teha suvalises järjekorras. See võimaldab viia sooritatavate opereerimiste arvu logaritmiliseks astendaja suhtes, kasutades "jaga ja valitse" tehnikat.

  Kirjutada funktsiooni `kõrgAste` uus variant lähtuvalt sellest tähelepanekust.
- Arvutada interaktiivses interpretaatoris funktsiooni `kõrgAste` abil lihtsal viisil avaldise

  $$\underbrace{\sin(\ldots(\sin}_{29} 100)\ldots)$$

  väärtus. Kasutada standardfunktsioone
  - infiksne `.` — funktsioonide kompositsioon (liitfunktsiooni moodustamine),
  - `id` — identsusfunktsioon (samasusfunktsioon).

- Lõpmatud listid.

  Haskellis võib täiesti korrektse ja mõtteka avaldise väärtuseks olla lõpmatu andmestruktuur, näiteks list. Selle teeb võimalikuks laisk väärtustamine. Iga listi struktuur ehitatakse mällu ainult niikaugele, kui programmi täitmise seisukohast tarvilik. Seega kui lõpmatul 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.

  Lihtsaim viis Haskellis lõpmatut listi tekitada on aritmeetilise progressiooni erisüntaksiga. Näiteks `[1 ..]` väärtuseks on lõpmatu list elementidega järgemööda 1, 2, 3 jne, `[1, 3 ..]` väärtuseks aga lõpmatu list elementidega järgemööda 1, 3, 5, 7 jne.

  Kontrollimiseks, kas avaldise väärtuseks on meie soovitav lõpmatu list, ei tasu seda avaldist lihtsalt niisama interaktiivse interpretaatori käsurealt ette anda, sest see tähendaks kogu lõpmatu listi täieliku väärtustamise nõuet ja programm jääks lõpmatult tööle. Selle asemel tasub rakendada avaldisele mõnd funktsiooni, mis arvutab lõpmatu listi järgi mõne teda iseloomustava lõplikult esituva väärtuse. Näiteks listi `[1 ..]` testimiseks võiks kästa väärtustada avaldis

  ```
  take 100 [1 ..].
  ```

  Selle avaldise väärtus on list 100 esimese elemendiga listist `[1 ..]`, mille täieliku väljaarvutamisega pole probleeme, mille vastus mahub tervikuna aknasse ja mille järgi võib üsna kindlalt oletada, et vaadeldav list väärtustatakse kuitahes kaugele õigesti. Kuna täielikult ei ole lõpmatut listi mingilgi tingimusel võimalik välja arvutada, on kuitahes kaugete elementide õigsus just see, mida lõpmatu listi korrektsuse all mõista saab.

  Teised lihtsad võimalused lõpmatuid liste defineerida on standardfunktsioonidega `repeat`, `cycle` ja `iterate`. 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. Funktsioon `iterate` võtab argumendiks funktsiooni f ja tema võimaliku argumendi x ning annab väärtuseks lõpmatu listi elementidega x, $f\ x$, $f\ (f\ x)$, $f\ (f\ (f\ x))$ jne.

  – Testida interaktiivse interpretaatori käsurealt senivaadeldud viisidel konstrueeritavaid lõpmatuid liste.

  Listi lõpmatuses pole Haskelli väärtustusmehhanismi jaoks midagi erilist. Rakendatakse neidsamu ümberkirjutusreegleid mis lõplikegi listide puhul. See on ka mõistetav, kuna süsteem ei saa ühegi listi kohta ju kindlaks teha, kas ta on lõpmatu või lõplik.

  Listidel töötavaid funktsioone defineerides tuleb mõelda ka selle peale, mis juhtub lõpmatu argumentlisti puhul. Funktsiooni töö sõnadessevalamisel enamasti piirdutakse töö iseloomustamisega lõplikel listidel. Lõpmatutel listidel võib seetõttu juhtuda üllatusi.

  Tihti teeb funktsioon lõpmatul listil siiski sedasama meile vajalikku tööd mis lõplikul listilgi. Näiteks kasutasime eelpool lõpmatust listist algusjupi kättesaamiseks funktsiooni `take`, seega `take` suudab ka lõpmatul listil võtta etteantud arvu esimesi elemente. See on seepärast, et esimese n elemendi arvutamiseks käseb `take` definitsioon (vt prelüüdist) otsida listist välja maksimaalselt n elementi ja unustab ülejäänud ära. Samas funktsioon `length`, mis lõplikul listil arvutab tema elementide arvu, jääb lõpmatu listi korral lõpmatult tööle, st meie oodatav käitumine `length` puhul ei laiene lõpmatutele argumentidele.

  – Mida teevad järgnevad standardfunktsioonid, kui mõni argumentlist on lõpmatu? Mõelda algul definitsiooni järgi ja seejärel testida interaktiivses interpretaatoris.

    ∗ `head`,
    ∗ `tail`,
    ∗ `null`,
    ∗ `length`,
    ∗ `++`,
    ∗ `!!`,
    ∗ `drop`,
    ∗ `takeWhile`,
    ∗ `dropWhile`,
    ∗ `filter`,
    ∗ `sum`,
    ∗ `reverse`.

  – Mida teeb funktsioon `concat`, kui

    ∗ argumentlisti mõni element on lõpmatu?
    ∗ argumentlist ise on lõpmatu?

    Kas `concat` tulemuslist võib tekitada ebameeldivaid üllatusi?

  Funktsioone kujul `map` f võib julgelt lõpmatutele listidele rakendada — tulemus vastab tavapärasele ootusele (argumentlisti igale elemendile rakendatakse funktsiooni f). Näiteks võib katseliselt kergesti veenduda, et `map (* 2) [1 ..]` väärtuseks on lõpmatu list elementidega 2, 4, 6, 8 jne.

  Esmapilgul võib tunduda, et funktsioonid kujul `foldr` $\oplus$ e ei anna lõpmatule listile rakendades kunagi midagi mõistlikku. Tekitab ju tema väärtustamine lõpmatul listil avaldise

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

  milles pole ühtki täielikult väärtustatud alamavaldist. See mulje on aga kahel põhjusel ekslik. Esiteks võib operaator $\oplus$ olla oma parema argumendi järgi laisk ja niipea, 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` $\oplus$ e lõpetab lõpmatul listil töö normaalselt lõpliku ajaga. Teiseks võib niisuguse avaldise väärtus olla ise lõpmatu list, mille mõned elemendid annab kätte juba $\oplus$ vasakpoolne argument. Niisuguse olukorra näiteks on juba vaadeldud funktsioonid `map` f, mis on defineeritavad kujul `foldr ((:) .` f`) []`.

– Lausearvutuse tehted `&&` ja `||` on oma parema argumendi järgi laisad. See tähendab `&&` puhul, et kui esimene argument on `False`, siis antakse `False` välja ilma paremat argumenti uurimata, ja `||` puhul, et kui esimene argument on `True`, siis antakse `True` välja ilma paremat argumenti uurimata. Sellest lähtuvalt leida operaator $\oplus$, väärtus e ja lõpmatu list l, mille korral `foldr` $\oplus$ e l lõpetab töö normaalselt lõpliku ajaga.

– Defineerida oma moodulis alternatiivne konjunktsioon või disjunktsioon, mis on laisk hoopis vasaku argumendi järgi. Veenduda, et standardse konjunktsiooni või disjunktsiooni asendamisel sellega eelmises punktis leitud näide enam ei tööta.

– Kas eelmises punktis defineeritud konjunktsiooni või disjunktsiooni korral võimaldab järsku `foldl` nüüd lõpmatul listil lõpliku ajaga vastuse kätte saada? Miks? Kas leiduvad operatsioon $\oplus$, element e ja lõpmatu list l, nii et `foldl` $\oplus$ e l omab mingitki mõtet?

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, ei jõuaks arvutus kunagi paari teise komponendini. Ometi on ta olemas, selles veendumiseks piisab lihtsalt võtta mõlemast komponentlistist esimene element: `(\ (xs , ys) -> (head xs , head ys))` ülaltoodud avaldisele rakendades annab tulemuseks

```
(1 , 2).
```

Element `2` tuli teisest listist.

– Defineerida muutuja, mille väärtuseks on lõpmatu list lõpmatutest listidest, ja rakendada interaktiivse interpretaatori käsurealt talle selline funktsioon, mis leiab neist igaühest mõned elemendid.

– Defineerida muutuja `ükskordüks`, mille väärtuseks on lõpmatu naturaalarvude korrutustabel.

– Defineerida funktsioon `nurk`, mis naturaalarvulisel argumendil n ja argumendil `ükskordüks` annab (lõpliku ajaga) vastuseks korrutustabeli ülemise vasakpoolse osa suurusega $n \times n$.

– Kas järgnevad Teie poolt varem praktikumis kirjutatud listifunktsioonid käituvad lõpmatutel listidel ootuspäraselt? Kui ei, anda neile uus definitsioon, mis töötab lõpmatutel listidel paremini.

  * `vahed`,
  * `kõikVõrdsed`,
  * `paarikaupaErinevad`.

- **Jätkuedastus**.

  Jätkuedastustehnika ideeks on arvutusprotsessi nägemine mitte kontekstivabana, vaid jätkuva protsessi algusosana.

  Niisiis on meil jätkuedastustehnika puhul tegemist kahe nähtusega ühe asemel: meid huvitav protsess ja tema jätk, mitte lihtsalt protsess. Mida täpselt kujutavad endast “arvutusprotsess” ja “jätk”, ei ole seejuures üheselt selge. Seda võib olenevalt vajadusest kodeerida mitmel moel.

  Siinkohal vaatleme vaid kaht käsitlusvarianti. Esiteks kõige klassikalisem käsitlus, kus nii protsessid kui jätkud on funktsioonid; teiseks variant, kus protsessid ja jätkud on listid, mida võib vaadelda kui vahetulemuste jadu.

  Jätkuedastusega on lahutamatult seotud kaks mõistet.

  Ühendamisoperatsioon. Tavalises jätkudeta käsitluses on protsess midagi, mis teisendab algoleku lõppolekuks. Jätkuedastustehnikas “edastatakse” protsessile ka tema jätk, kusjuures protsess koos oma jätkuga moodustab kokku samuti jätku, mida võib omakorda edastada eelmisele protsessile jne. Seda uue jätku moodustamise operatsiooni protsessi ja tema jätku põhjal me nimetamegi ühendamiseks. Seega jätkuedastustehnikas on protsess midagi, mis teisendab enda jätku eelmise protsessi jätkuks.

  Vaatame olukorda, kus protsessid ja jätkud on funktsioonid. Olgu P protsess ja C tema jätk; kuna C peab jätkama sealt, kus P lõpetab, siis tüübid peavad olema P `::` a `->` b ja C `::` b `->` o mingite a, b, o korral. Protsessi P ja jätku C ühendamisel saadav jätk on funktsioon, mis oma argumendile rakendab kõigepealt P-d ja tulemusele C-d. Niisiis sellisel juhul on ühendamisoperatsiooniks funktsioonide kompositsioon.

  Kui protsessid ja jätkud on n-ö vahetulemuste listid, siis peavad nende elemendid olema sama tüüpi, sest protsessi P ja jätku C põhjal koostatud ühendatud jätk on list, mille alguses on P elemendid ja seejärel C elemendid. Ühendamisoperatsiooniks on seega listide konkatenatsioon.

  Lõppjätk. Kui iga jätk oleks vaid mingi protsessi ja mingi teise jätku ühend, oleksid kõik meie protsessid põhimõtteliselt lõpmatud. Seega on lõpetamiseks vaja mingit erilist jätku, mis ei sisalda endas enam mitte mingit tegevust ja mille edastamisel mistahes protsessile saame ühendamisel tulemuseks jätku, mis seisneb selle protsessi täitmises ja koheses lõpetamises selle järel.

  Funktsioonide korral sobib lõppjätkuks identsusfunktsioon `id`. Listide korral sobib lõppjätkuks tühi list `[]`.

Jätkuedastusstiilis programmeerimine tähendabki protsesside programmeerimist funktsioonidena nende jätkudest eelmise protsessi jätkudeks, st sisuliselt P asemel jätkude P-ga ühendamise operatsiooni P^{C} programmeerimist. Funktsioonide puhul niisiis programmeeritakse

P `::` a `->` b asemel P^{C} `::` $(b$ `->` $o)$ `->` $(a$ `->` $o)$ ning listide korral P `::` `[`a`]` asemel P^{C} `::` `[`a`]` `->` `[`a`]`. Protsessi P saame pärast defineerida kui P^{C} lõppjätkule rakendamise tulemuse.

Vaatame triviaalseid näiteid.

1. Tahame defineerida funktsiooni $f(n) = 2n - 1$. Jätkuedastustehnikas programmeerime f^{C}:

   ```
   fC c n = c (2 * n - 1).
   ```

   See tähendab, `fC` võtab argumendiks jätku `c`; et tulemuseks on omakorda funktsioon, võtab ta veel teise argumendi `n` ning teostab kõigepealt meid huvitava operatsiooni (korrutab 2-ga ja lahutab 1) ja seejärel jätkab jätkuga `c`.

   Nüüd saame soovi korral f kätte, defineerides

   ```
   f = fC id.
   ```

2. Tahame defineerida listi, mille elementideks on kõik numbrid 0-st 9-ni. Jätkuedastustehnikas kirjutame

   ```
   numbridC c = 0 : 1 : 2 : 3 : 4 : 5 : 6 : 7 : 8 : 9 : c.
   ```

   Siin jätk `c` on list, mis peab jääma tulemuslisti lõppu — seda antud definitsioon ka tagab. Pangem tähele, et meid algselt huvitanud list selles definitsioonis osaavaldisena ei esine, st oleme defineerinud kogujätku otse, ilma meid huvitavale algusosale viitamata. See on väärtuslik tähelepanek, sest kui jätkuedastustehnikas programmeerimine nõuaks alati ka protsesside endi programmeerimist sellisena nagu ilma jätkutagi, siis ei oleks ju jätkuedastustehnikal mõtet.

   Soovi korral aga saame algselt huvitanud listi kätte, defineerides

   ```
   numbrid = numbridC [].
   ```

– Realiseerida ise kas need või mingid muud lihtsad näited jätkuedastustehnikaga. Testides jätke teisendavat funktsiooni, andke lisaks lõppjätkule ka teisi võimalikke jätke argumendiks.

Funktsiooniliste või listiliste jätkude puhul tulevad jätkuedastustehnika võlud välja alles rekursiivsete definitsioonide korral. Et rekursiivne definitsioon viib iteratiivse arvutuseni, tekib niisuguse definitsiooni järgi arvutades järgmine protsess:

$$s_0 \to s_1 \to s_2 \to \ldots \to s_{n-1} \to s_n \to c,$$

kus $s_0, s_1, \ldots, s_n$ on meid huvitava funktsiooni arvutamise vahetulemused ja c sellele järgnev võimalik jätk. Jätkuedastustehnikas kajastub sama protsess noolte ümberpööratud suunaga:

$$d_0 \leftarrow d_1 \leftarrow d_2 \leftarrow \ldots \leftarrow d_{n-1} \leftarrow d_n \leftarrow c,$$

st jätku c järgi leitakse jätk d_n (lisades eelneva protsessi viimase lüli c ette), d_n järgi d_{n-1} jne, kuni lõpuks d_0 on jätk, mis sisaldab endas algse funktsiooni arvutuse kõik lülid, millele järgneb c.

Vaatame, kuidas kukub välja faktoriaali defineerimine jätkuedastustehnikaga.

```
factC c n
  = case compare n 0 of
      GT
        -> factC (c . (n *)) (n - 1)
      EQ
        -> c 1
      _
        -> error "factC: negatiivne argument"

fact = factC id
```

Funktsioon `factC` saab võrreldes tavalise faktoriaaliga lisaargumendina jätku `c`. Kui teine argument `n` on 0 (*case*-avaldise teine variant), siis faktoriaaliarvutuse tulemus on triviaalselt 1, seega koos jätkuga `c` on lõpptulemus lihtsalt `c 1`. Kui aga `n` on 0-st suurem (*case*-avaldise esimene variant), siis arutleme nii: see, kui arvutada kõigepealt `n` faktoriaal ja siis rakendada tulemusele `c`, on samaväärne sellega, kui kõigepealt arvutatakse $(\mathtt{n} - 1)!$ ja siis rakendatakse tulemusele `n`-ga korrutamise ja `c` kompositsioon.

Tähelepanelik lugeja kindlasti märkab sarnasust akumulaatoritehnikaga — tõepoolest, jätkuargument on siin akumulaator, tegemist on **akumuleeruva jätkuga**.

– Mängida samm-sammult läbi avaldise `factC c 3` väärtustusprotsess ja veenduda, et `c` lõppargumendiks kujuneb tõesti $3!$ nagu vaja.

Näidet rekursiivse protsessi defineerimisest listiliste jätkudega aga pole tarvis kaugelt otsida. Lk 13 on toodud funktsiooni `reverse` akumulaatoriga definitsioon, kus akumulaator pole õigupoolest midagi muud kui akumuleeruv jätk.

See on ka esimeseks näiteks sellest, kuidas jätkuedastustehnika võib aidata kaasa efektiivsuse tõstmisele.

– Kõrvutada lk 13 antud `reverse` definitsiooni lokaalse abifunktsiooni ning `factC` definitsioone. Kas pole sarnased?

– Lk 16 on defineeritud `qsort` "jaga ja valitse" tehnikaga. Paraku pole ka see kõige efektiivsem. Põhjus on selles, et iga konkatenatsioon tõstab vasakpoolse argumentlisti elemendid ükshaaval parempoolse argumentlisti ette. Viidatud definitsiooni terminites siis `qsort us` kõik elemendid tõstetakse pärast sorteerimist ükshaaval ümber listi `x : qsort vs` ette. Kuna `qsort us` ise on sama algoritmi töö tulemus, siis osa sellest listist on järelikult juba tema koostamise käigus varem ümber tõstetud, need elemendid siis tõstetakse kokkuvõttes ümber kaks korda. Sama arutelu jätkates saab selgeks, et mida väiksemad algse listi elemendid, seda rohkem kordi nad satuvad jaotamisel vasakpoolsesse ossa ning niisama palju kordi tõstetakse neid tühjalt (veel pärast ärasorteerimist) ümber.

Realiseerida "jaga ja valitse" kiirsorteerimisalgoritm akumuleeruva jätkuga, nii et ühtki elementi ei tõstetaks liigselt ringi.

Vaatame veel kaht näidet jätkuedastustehnika kasutamisest funktsiooniliste jätkude korral.

Olgu meil tarvis kirjutada funktsioon `eraldaViimane`, mis võtab argumendiks mingi listi l ja kui see on lõplik ja mittetühi, siis annab tulemuseks paari, mille esimene komponent on listi l viimane element ja teiseks komponendiks listi l ülejäänud elementide list elementidega samas järjekorras.

Enne jätkuedastust õpitud tehnikate abil võiksime seda realiseerida järgneval kahel viisil.

```
eraldaViimane1 (x : xs)
  = let
        eraldaViimane x (y : ys)
          = (\ (z , us) -> (z , x : us))
              (eraldaViimane y ys)
        eraldaViimane x _
          = (x , [])
      in
      eraldaViimane x xs
eraldaViimane1 _
  = error "eraldaViimane1: tühi list"

eraldaViimane2 (x : xs)
  = let
        eraldaViimane as x (y : ys)
          = eraldaViimane (x : as) y ys
        eraldaViimane as x _
          = (x , reverse as)
      in
      eraldaViimane [] x xs
eraldaViimane2 _
  = error "eraldaViimane2: tühi list"
```

Esimene on otsene realisatsioon, kus defineeritava funktsiooni väärtus listil avaldatakse tema väärtuse kaudu sabal. Teine on tavalise akumulaatoriga realisatsioon. Kui esimese realisatsiooni puuduseks on, et igal rekursioonisammul konstrueeritakse vaheandmestruktuur — paar, mida hiljem vaja ei tule —, siis teise realisatsiooni puuduseks on, et elemendid akumuleeruvad listi vales järjekorras ja akumulaator tuleb enne lõpliku vastuse väljaandmist

ümber pöörata. Seda ümberpööramise lisandumist on ilmselt kogenud kõik, kes asju listiakumulaatoriga on defineerinud.

Esimest definitsiooni otse jätkuedastusse ümber pannes saaksime järgmise variandi.

```
eraldaViimaneC c (x : xs)
  = let
        eraldaViimane c x (y : ys)
          = eraldaViimane (
              c . (\ (z , us) -> (z , x : us))
            ) y ys
        eraldaViimane c x _
          = c (x , [])
      in
      eraldaViimane c x xs
eraldaViimaneC _ _
  = error "eraldaViimaneC: tühi list"

eraldaViimane3 = eraldaViimaneC id
```

Näeme, et kuigi vastuslisti arvutatakse akumulaatoris, jäävad seekord elemendid listi kohe õiges järjekorras, nii et lõpus pole ümberpööramist vaja. Samas toimub jällegi paaride konstrueerimine igal rekursioonisammul. Sellest saame lahti, kui asendame paare argumendiks võtvad jätkud funktsioonidega, mis võtavad argumente järgemööda, st vastavate funktsioonidega *curried* kujul. Funktsioon

```
c . (\ (z , us) -> (z , x : us)),
```

mis on sama mis

```
(\ (z , us) -> c (z , x : us)),
```

asendub funktsiooniga

```
(\ z us -> c z (x : us)).
```

Lõppjätk peab nüüd eraldatud argumendid paariks kokku panema, st `id` asemel tuleb `(,)` (funktsioon, mis võtab kaks argumenti ja annab välja nende paari).

Saame definitsiooni

```
eraldaViimaneC' c (x : xs)
  = let
       eraldaViimane c x (y : ys)
         = eraldaViimane (\ z us -> c z (x : us)) y ys
       eraldaViimane c x _
         = c x []
    in
    eraldaViimane c x xs
eraldaViimaneC' _ _
  = error "eraldaViimaneC': tühi list"

eraldaViimane4 = eraldaViimaneC' (,)
```

Üheks jätkuedastustehnika plussiks on ülesande lahendamine abstraktsemal tasemel, mis tihti annab tasuks ühekorraga selle ja mitmete teiste meid huvitavate ülesannete lahendusi. Näiteks olles defineerinud `eraldaViimaneC'` nagu viimati, saame argumendiks antavaid jätke modifitseerides näiteks järgmised kasulikud funktsioonid:

```
viimane          = eraldaViimaneC' const
viskaViimaneÄra  = eraldaViimaneC' (flip const)
tooViimaneEtte   = eraldaViimaneC' (:)
```

(Siin `flip` on standardfunktsioon, mis vahetab oma argumentfunkstiooni argumentide järjekorra.)

Kui funktsiooni `eraldaViimane` ilma jätkudeta programmeerida, on otsitavad kaks suurust lõpus meie käsutuses kindlasti paarina. Jätkude kasutamise korral jääb see lahtiseks, me võime nendega mida iganes teha, neist näiteks hoopis listi moodustada nagu `tooViimaneEtte` korral.

Teise näitena olgu meil vaja defineerida funktsioon `span`, mis võtab argumentideks predikaadi p ja listi l ning annab tulemuseks paari, mille esimeseks komponendiks on listi l elemendid algusest kuni esimese tingimuse p mitte rahuldava elemendini (see väljaarvatud) ning teiseks komponendiks järelejääv sabaosa. (`span` on tegelikult standardfunktsioon, prelüüdis olemas.)

Jätkuedastustehnikaga, kus paarifunktsioonid *curried* kujule viidud, saame järgmise definitsiooni.

```
spanC c p zs@ ~(x : xs)
  | null zs || not (p x)
    = c [] zs
  | otherwise
    = spanC (c . (x :)) p xs
```



```
span = spanC (,)
```

Siin esimeses reas olev näidis `~(x : xs)` on laisaks forsseeritud listinäidis. Ilma tildeta ees tähendaks selline näidis, et deklaratsioon kehtib ainult nende listide jaoks, mis selle näidisega sobituvad, st mittetühjade listide jaoks. Kuna muid deklaratsioone pole, annaks `spanC` sellisel juhul tühjal listil veateate. Tilde aga teeb väärtustaja selle näidise ees n-ö pimedaks, nii et ta ei uuri tegeliku argumendi vastavust sellele näidisele enne, kui seal olevaid muutujaid reaalselt vaja läheb. Lühidalt öeldes, `~(x : xs)` tähistab suvalist listi, ka tühja, aga kui siiski jopastab olema mittetühi list, siis `x` tähistab tema pead ja `xs` saba.

Selliste näidistega programmeerimisel tuleb olla tähelepanelik, sest kui tegelikult tuleb argumendiks tühi list ja väärtustamisel küsitakse järsku tema pead ja saba, siis tekib täitmisaegne viga. Antud definitsiooni korral seda ei juhtu, sest alati, kui list on tühi, annab esimese valvuri disjunktsiooni vasakpoolne argument `True`, misjärel disjunktsiooni teist poolt ei uurita ning võrdusmärgist paremal peale ega sabale ei viidata.

Selles definitsioonis aitab laisa näidise kasutamine koodiridu kokku hoida, sest parem pool `c [] zs` on ühine tühja listi juhule ja mittetühja listi juhule, kus pea predikaati ei rahulda.

- Millised standardfunktsioonid oskate defineerida `spanC` abil, varieerides jätku?
- Kirjutada funktsioon `splitAtC`, mis realiseerib prelüüdis olemasoleva funktsiooni `splitAt` jätkuedastustehnikaga. Millised standardfunktsioonid saab defineerida siin jätku varieerides?
- Realiseerida jätkuedastustehnikaga funktsioon `span2`, mis võtab argumentideks binaarse predikaadi p ja listi l ning annab tulemuseks paari, mille esimeseks komponendiks on listi l algusosa elemendid kuni esimese kohani, kus predikaat p kahel järjestikusel elemendil ei kehti, ja teiseks komponendiks järelejääv sabaosa.
- Realiseerida Fibonacci arve arvutav funktsioon `fib` akumuleeruva jätkuga.
- Eelmise ülesande lahenduse abil kirjutada funktsioon, mis argumendil n leiab n-nda ja $(n+1)$-se Fibonacci arvu korrutise.

Jätkuedastustehnika puuduseks on, et funktsiooniliste akumuleeruvate jätkude korral pole võimalik vahetulemusi jooksvalt väärtustada. Kui tegemist on listidel töötava ja liste genereeriva funktsiooniga, võib kergesti juhtuda, et lõpmatu argumentlisti korral tekib lõpmatu tsükkel, mis ei suuda vastusest ühtki elementi välja anda, kuigi ilma jätkudeta sama asja realiseerides saaksime vastuslisti elemendid kätte.

- Kas moodulis $\mathrm{Data.List}$ olevat funktsiooni `partition` saab samaväärselt defineerida jätkuedastustehnikaga?
- Kas Teie poolt kirjutatud funktsioonid `splitAtC` ja `span2C` ikka töötavad lõpmatul listil?

- Kirjutada funktsioon `posC`, mis realiseerib jätkuedastustehnikaga listist esimese predikaati rahuldava elemendi järjekorranumbri leidmise. Kui list on lõplik ja ükski element predikaati ei rahulda, on vastuseks -1.
- Kasutades siiani defineeritud jätkuga funktsioone `fC`, `factC` ja `posC`, kirjutada funktsioon `kolmKoos`, mis võtab argumendiks predikaadi p ja listi l, leiab listist l esimese predikaati rahuldava elemendi järjekorranumbri, leiab selle järjekorranumbri faktoriaali ning korrutab selle tulemuse 2-ga ning lahutab 1. Kui list on lõplik ja ükski element predikaati ei rahulda, on vastuseks -1.

Eeldefineeritud funktsioonide kasutamine

Funktsionaalse programmeerimise üks suuri eesmärke on kõrge abstraktsioonitase. Eelnevas nägime, et funktsioone on võimalik anda teise funktsiooni argumendiks, seega on võimalik defineerida käitumist parameetrilisena teise käitumise suhtes. Niisuguselt defineeritud funktsioonid on laiema kasutusega kui funktsioonid, mille argumentide hulgas funktsioone pole, ja võimaldab vältida koodi duplitseerimist (enam-vähem sama käitumise programmeerimist ikka ja jälle uuesti).

Paljud preluudis defineeritud funktsioonid on mõeldud selliseks universaalseks kasutamiseks. Kui konkreetne ülesanne lahendub erijuhuna mõnest universaalsest funktsioonist, on halb stiil minna sellest mööda. Universaalsete funktsioonide õige kasutamine tagab koodi lugejale kiirema arusaamise programmist.

Lisaks tuleks võimalusel vältida tarbetute argumentide ladumist definitsiooni. Seda nn *point-free* stiili me oleme juba näinud nt faktoriaalipere funktsioonide definitsioonides lk 12, kus `fact n = kõrgFact (*) 1 n` asemel on lihtsalt `fact = kõrgFact (*) 1` — kui defineeritava funktsiooni töö seisneb lihtsalt oma argumendi ettesöötmises mingile teisele funktsioonile, võib selle argumendi mõlemalt poolt võrdusmärki ära "taandada". Samuti näeme argumendivaba stiili funktsiooni `reverse` definitsioonis lk 13, kus argumentlist, mida ümber pööratakse, on jäetud kirjutamata.

Teoreetiliselt võib sellise võtte võimalikkust põhjendada matemaatilise kokkuleppega, et funktsioonid f ja g on võrdsed, kui iga argumendi x korral $f(x) = g(x)$. Seda omadust nimetatakse **ekstensionaalsuseks**. Haskellis peab see seadus paika alati, kui kumbki funktsioonidest f ja g pole vigane. See tähendab, et see seadus ei kehti ainult juhul, kui üks funktsioonidest f ja g on vigane, teine aga ise pole vigane, kuid annab igal oma argumendil vigase tulemuse. Niisugune erijuht ei paku enamasti mingit huvi ja igapäevaprogrammeerimises seega võib ekstensionaalsust eeldada.

- Funktsioon `flip`.

  Funktsioon `flip` vahetab ära oma argumentfunktsiooni argumentide järjekorra. Näiteks kui on vaja defineerida funktsioon, mis töötab nagu astendamine, kuid võtab astendaja enne astme

  alust, on seda kõige mugavam ja kergemini loetavam teha funktsiooniga `flip`, kasutades argumendivaba stiili:

  ```
  minuAste = flip (^).
  ```

  - Standardfunktsioon `const` võtab järjest kaks argumenti ja annab tulemuseks neist esimese. Kirjutada argumendivabas stiilis funktsioon, mis võtab järjest kaks argumenti ja annab välja neist teise.
  - Kirjutada argumendivabas stiilis funktsioon `jäägid`, mis argumendil n annab välja listi komponentidega $0, 1, \ldots, n-1$.

  Kui "flipitava" funktsiooni üks argument on antud, võib `flip` asemel kasutada parempoolset sektsiooni. (Kompilaator tegelikult kirjutabki parempoolsed sektsioonid ümber funktsiooni `flip` kaudu, tuum-Haskellis neid ei ole.) Näiteks `flip (^) 2` on funktsioon, mis astendab oma argumenti 2-ga, st sama mis `(^ 2)`.

  - Kirjutada eelmise ülesande funktsioon (argumendil n annab listi elementidega $0, 1, \ldots, n-1$) argumendivabas stiilis, kuid `flip` asemel parempoolse sektsiooniga.

- Funktsioon `..`

  Kui funktsiooni käitumine seisneb oma argumendile mingite funktsioonide järjest rakendamises, kusjuures need funktsioonid sellest argumendist ei sõltu, võib kasutada kompositsiooni.

  - Kirjutada funktsioon `takeLõpust`, mis töötab nagu `take`, ainult et võtab elemente listi lõpust. Ühest argumendist lahtisaamiseks kasutada kompositsiooni.

- Funktsioon `map`.

  Kui vaja on näiteks funktsiooni, mis võtab argumendiks listi ja annab tulemuseks sellest listist kõigi elementide kahekordistamise teel saadud listi, ei peaks mitte rekursiooniskeemi programmeerima hakkama, vaid kasutama standardfunktsiooni `map`. Definitsioon tuleks

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

  Nii lihtsa definitsiooni puhul tasuks kaaluda lausa mõtet uue muutuja `double` kasutuselevõtust loobuda ja, igal pool kus vaja, kirjutada lihtsalt `map (* 2)` sisse.

  - Kirjutada argumendivabas stiilis funktsioon `pealk`, mis teisendab oma argumentstringi kõik tähed suureks.

- Funktsioonid `concat` ja `>>=`.

  Mõnikord pole `map` rakendamise tulemus otseselt see, mida ülesanne nõudis, nii et tulemust peab veel töötlema. See pole argument temast loobumise kasuks. Tihti juhtub nii, kui listi iga elemendiga on põhimõtteliselt vaja sooritada operatsioon, mille tulemus on ise list, kuid funktsiooni spetsifikatsiooniga listide list ei sobi. Siis tuleb saadud listid töö lõpuks üheks konkateneerida.

  Näiteks olgu vaja kirjutada funktsioon `pikendaVahed`, mis asendab stringis tühikud kahekordse tühikuga, jättes muud sümbolid paigale. Kuna kahekordne tühik pole sümbol, vaid string, mis on sümbolist erinev tüüp, siis pole võimalik funktsiooni `pikendaVahed` esitada kujul `map` f, sest f argumenttüüp peaks olema `Char` (string on nimelt sellist tüüpi objektide list), väärtustüüp aga `String`, seega tulemus oleks juba stringide, mitte sümbolite list.

  Küll aga saame soovitud tulemuse, kui me lõpus selle stringide listi komponendid konkateneerime. Õige definitsioon on näiteks

  ```
  pikendaVahed
    = let
        f c
          = if c == ' ' then "  " else [c]
      in
      concat . map f
  ```

  Konstruktsioon kujul `concat . map` f on nii levinud, et prelüüdis on defineeritud spetsiaalne infiksoperaator `>>=` deklaratsiooniga

  ```
  xs >>= f
    = concat (map f xs)
  ```

  - Defineerida funktsioon `pikendaVahed` infiksoperaatori `>>=` abil.
  - Kirjutada `>>=` abil funktsioon `toDOS`, mis asendab argumentstringis esinevad sümbolid `\n` (reavahetused) DOSi kodeeringu reavahetustega, milleks on kahesümboliline järjend `\r\n`. (Haskelli süntaksis on sümbolite kirjakujud `\n` ja `\r` olemas.)

- Funktsioon `foldr`.

  Funktsioon `foldr` on defineeritud seostega

  ```
  foldr op e (x : xs)
    = op x (foldr op e xs)
  foldr op e _
    = e
  ```

  See tähendab, et kujul `foldr` f e esituvad ainult sellised funktsioonid, mille käitumine listil on määratud listi pea ja sama funktsiooni töö tulemusega listi sabal.

  Loengumaterjalis on toodud palju näiteid funktsioonide definitsioonidest kujul `foldr` f e. Samas leidub palju funktsioone, mis niisugusel kujul ei esitu; sellised on nt `dropWhile` p, kuna vajavad vähemalt mõnel rekursioonisammul lisaks rekursiivse töö tulemusele ka listi saba ennast.

  - Kuivõrd on kujul `foldr` f e väljendatavad
    * `head`, `tail`, `null`, `filter` p, `takeWhile` p?
    * Teie poolt praktikumis kirjutatud funktsioonid `vahed`, `kõikVõrdsed`, `paarikaupaErinevad`?

    Väljendada, mis võimalik.

Sisend-väljund

Sisend-väljundi alla kuulub igasugune programmi suhtlemine keskkonnaga. Seni oleme saanud sellest mööda vaadata, sest senikäsitletud programmide keskkonnaga suhtlemise võimalikud tulemused on olnud triviaalsed: kas mõju avaldiste väärtustamise käigule puudub või tekib ületamatu viga, mille tagajärjel programmi töö katkeb (nt mälu täitumise korral).

Reaalsete ülesannete lahendamisel sellest loomulikult ei piisa. On vaja, et programmid suudaks aktiivselt suhelda keskkonnaga (sh inimesest kasutaja, failisüsteemi, juhuarvude generaatori ja kõikvõimalike teiste programmidega) ja seada oma käitumist sõltuvalt keskkonnalt saadud signaalidest.

Igasugune mittetriviaalne mõju keskkonnalt on olnud seniste programmide puhul põhimõtteliselt võimatu, sest avaldiste teisendamise käik on olnud programmi tekstiga (ja interaktiivse interpretaatori käsurealt antud avaldisega) üheselt määratud. Me pole õppinud veel ühtki funktsiooni, mis tooks sisse keskkonnamõju.

Funktsionaalses programmeerimises on niisuguste mõjude sissetoomine põhimõtteliselt raske ilmutatud viidatavuse omaduse nõude tõttu, millest tulenevalt muutujate väärtused töö käigus ei muutu. Põhimõtteliselt pole võimalik näiteks muutuja `getChar :: Char`, mille väärtuseks oleks kasutaja sisestatav sümbol ja mille ootamiseks programmi töö iga kord selle muutuja esiletulekul peatub.

Et keskkonnamõju arvestavate programmide kirjutamist siiski funktsionaalse paradigma puhtust säilitades võimaldada, on Haskellis tehtud järgmist.

1. Keskkonnaga suhtlemisega seotud avaldiste jaoks on defineeritud omaette tüübid `IO` a, kus a on suvaline tüüp. Näiteks Haskelli prelüüdis on tõesti olemas muutuja `getChar`, kuid tema tüüp pole mitte `Char`, vaid `IO Char`.

   Tüüpi `IO` a omavate avaldiste väärtuseks on **tegevus** (ingl action), mis **edastab** programmis järgnevale tegevusele (mis see ka poleks) ühe a-tüüpi väärtuse. Kust ta selle väärtuse võtab, on tema definitsiooni või realisatsiooni küsimus; kui ta saab selle keskkonnalt, tekitab ta järgneva programmiosa sõltuvuse sellest keskkonnasignaalist.

Näiteks `getChar` küsib standardsisendist sümboli ja edastab selle järgnevale tegevusele, seega programmi järgnev tegevus võib kasutaja sisestatud sümbolist sõltuda (kuna edaspidi saab programm seda sümbolit kasutada).

Niisuguse tõlgenduse puhul jääb ilmutatud viidatavus püsima, sest muutuja `getChar` väärtuseks olev tegevus on abstraktselt sama, sõltumata sellest, millise sümboli kasutaja tegelikult sisestab.

Täpsem olles tuleb märkida, et tegevused tüüpi `IO` a võivad loomuldasa ka ebaõnnestuda (keskkonnaga suhtlemisel on see tavaline, nt kui nõutud nimega faili ei ole), mispuhul a-tüüpi väärtuse edastamise asemel tegevus hoopiski **heidab** erindi tüübist `IOError`.

Lisaks defineeritakse mõned primitiivtegevused ning operatsioonid uute tegevuste konstrueerimiseks, näiteks kahe tegevuse järjest sooritamise operatsioon, erindi püüdmise operatsioon jm. Neid on kirjeldatud pisut hiljem.

2. Tüübid `IO` a on tehtud eriliseks selles mõttes, et niisugust tüüpi avaldiste väärtustamisel lisaks väärtustamisele see väärtuseks olev tegevus ka sooritatakse. See on midagi põhimõtteliselt uut lisaks senisele.

   Ei oleks ju mingi probleem Haskelli võimsa tüübisüsteemi juures eelmises punktis kirjeldatud tüübid ja primitiivoperatsioonid endal defineerida. Kuid omadefineeritud tüüpide puhul oleksid nad "elutud", sest nende väärtustamine viiks nad parimal juhul mingile kanoonilisele kujule ja loeks töö sellega lõppenuks — näiteks `getChar` viidaks väärtustamise käigus kujule, mis väljendaks mingil viisil sümboli küsimist kasutajalt, avaldaks selle tegevuse mingite atomaarsete operatsioonide kaudu (interaktiivne interpretaator kirjutaks lõpus välja selle kanoonilise kuju), kuid tegevus ise — sümboli küsimine — jääks sooritamata.

Paljud keskkonnaga suhtlevad operatsioonid ei tekita õnnestumise korral sõltuvust ühestki uuest väärtusest. Sellised on kõik operatsioonid, mis ainult saadavad signaale välja, aga ei töötle saadavaid — st mis ainult "kirjutavad", aga ei "loe". Niisugusel juhul, kui pole ka põhjust mingit juba olemasolevat väärtust edastada, loetakse tegevus olevaks tüüpi `IO ()`, kus `()` on **ühiktüüp**, mille ainsaks normaalseks väärtuseks on tühi kirje `()`. Niisiis sellised tegevused edastavad väärtuse `()`.

- Tutvuda loengumaterjalis toodud sisend-väljundiga seotud standardoperatsioonidega. Proovida operatsioone
  - `putChar`,
  - `putStr`,
  - `putStrLn`,
  - `print`,
  - `getChar`,
  - `getLine`

  kasutada interaktiivse interpretaatori käsurealt.

Pangem järgnevas tähele ja arvestagem, et tegevuste poolt edastatavaid väärtusi on võimalik kasutada ainult järgnevate tegevuste abil. Ei ole põhimõtteliseltki võimalik kirjutada funktsioone tüübiga `IO` a `->` a, mis argumendil a annaksid väärtuseks tegevuse a poolt edastatava väärtuse. Selline funktsioon rikuks ilmutatud viidatavuse omadust, sest sama tegevus võib erinevatel kordadel edastada erinevaid väärtusi.

Põhilisimad tegevusi kombineerivad operatsioonid on järgmised kaks.

1. Funktsiooni `return ::` a `-> IO` a väärtuseks argumendil x on väärtuse x edastamine (st tegevus, mis sisaldab parajasti väärtuse x edastamist).

   Seega kui imperatiivsetes programmeerimiskeeltes tihti esinevat `return`-käsku nimetatakse tagastuseks, siis siinset operatsiooni `return` tuleks nimetada **edastuseks**. Käitumised on põhimõtteliselt erinevad: pangem tähele, et Haskelli `return` ei hüppa alamprogrammist välja. Nimi `return` on Haskellis sellele operatsioonile valitud vahest sellepärast, et reaalsete programmeerimisülesannete puhul kipub alamprogrammide viimaseks tegevuseks olema `return` x, kus x võtab kokku need alamprogrammis kasutatud väärtused, mida ka edaspidi tahetakse kasutada, unustades ära ülejäänud.

2. Operatsioon `(>>=) :: IO` a `->` $(a$ `-> IO` $b)$ `-> IO` b kombineerib kaks tegevust üheks. Kui tegevus a edastab väärtuse x, siis tegevus a `>>=` k koosneb tegevusest a, millele vahetult järgneb tegevus $k\ x$; see tähendab muuhulgas, et a `>>=` k edastab või heidab sama väärtuse, mida vastavalt edastab või heidab $k\ x$. Kui aga tegevus a heidab erindi e, siis a `>>=` k tegevus on sama mis a-l; see tähendab, et funktsioonist k sel juhul midagi ei sõltu ning ka a `>>=` k heidab sama erindi e.

   Operatsiooni >>= nimetatakse **sidumiseks** (ingl bind).

   Näeme, et nipp, mis võimaldab tegevuste poolt edastatavaid väärtusi kasutada, on nende väärtuste sundsuunamine funktsiooni parameetriks. Ilmutatud viidatavust ei rikuta, kuna funktsiooni parameeter niikuinii algväärtustatakse igal väljakutsel uuesti ja võib võtta erinevatel kordadel erinevaid väärtusi.

Standardfunktsiooni `putChar :: Char -> IO ()` väärtuseks argumendil c on tegevus, mis saadab standardväljundisse sümboli c. Niisiis avaldise

```
getChar >>= putChar . toUpper
```

väärtuseks on tegevus, mis koosneb standardsisendist sümboli ootamisest ja saadud sümbolile vastava suurtähe saatmisest standardväljundisse.

Paneme tähele, et `putChar` c ei edasta järgnevale tegevusele sümbolit c, mistõttu ka toodud näiteprogramm "unustab" selle sümboli, mille ta sisse luges. Kui tahaksime, et meie tegevusblokk edastaks ka selle sümboli, peaksime kirjutama nii:

```
getChar >>= \ c -> putChar (toUpper c) >> return c
```

Siin `(>>) :: IO` a `-> IO` b `-> IO` b on standardfunktsioon, mis on defineeritud deklaratsiooniga

```
t >> u = t >>= const u,
```

st tegemist on tegevuste järgnevusega, kus teine tegevus esimese edastatavast väärtusest ei sõltu.

Mugavuse mõttes on Haskellis olemas erisüntaks niisuguste programmide kirjutamiseks, n-ö *do*-avaldised. Ülaltoodud avaldis näeks *do*-avaldisena samaväärselt ümber kirjutatuna välja kujul

```
do
  c <- getChar
  putChar (toUpper c) .
  return c
```

Järgnevate ülesannete lahendamiseks teha uus kataloog `IO`, mille alla koondate kõik selle teema ülesannete lahendused, igaüks eraldi failis.

Programmid peavad olema käivitatavad otse käsurealt ilma interaktiivse interpretaatori abita (testida võib muidugi interaktiivses interpretaatoris). Selleks tuleb moodulis defineerida muutuja `main`, mille väärtuseks on defineeritud soovitud tegevus. Moodulinimi peab olema Main.

Programmide käivitamiseks on kaks võimalust.

1. Programmiga `runhugs` käivitamiseks ülikooli failisüsteemil töötavates Unix-masinates tuleb failile anda käivitusõigus ja panna tema esimeseks reaks `#!/usr/local/bin/runhugs` (see rida ütleb *shell*ile, millisele programmile tuleb fail argumendiks anda). Seejärel jookseb Haskell-programm otse käsurealt, kui anda käsuna programmifaili nimi.

2. Programmi p`.hs` või p`.lhs` kompileerimiseks masinkoodi programmiga `ghc` tuleb anda käsk

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

   Seejärel käivitada tekkinud programm q. (Normaalne on võtta $q = p$.)

- Kirjutada programm `Hello`, mis kirjutab ekraanile "Hello, world!", Teie nime vms. Jälgida, et käsurealt käivitamisel tuleks viip uuele reale, mitte programmi poolt kirjutatud teksti otsa.

- Kirjutada programm `Symbolid`, mis ootab klaviatuurilt sümboleid ja igaühe kohta ükshaaval diagnoosib, kas tegu on väiketähega, suurtähega, numbriga või millegi muuga. Programm lõpetab töö, kui sisestatakse *enter*. Jälgida, et lõpu-*enter* ei tekitaks asjatut tühja rida.

- Kirjutada programmid `Koodid1` ja `Koodid2`, mis ootavad klaviatuurilt sümboleid, kuid ekraanile toovad hoopis sümbolite koodid (sümboleid mitte), igaühe eraldi reale. Programm `Koodid1` lõpetab töö, kui sisestatakse sama sümbolit kaks korda järjest; programm `Koodid2` lõpetab töö, kui sisestatakse sümbol, mida on programmi töö jooksul juba sisestatud, ning annab lõpetamisel teate kordumise kohta.

  Mõlemal juhul jälgida, et sümboli kordumine identifitseeritaks kohe peale tema sisestamist ja tavapärane reaktsioon (koodi ekraanilekirjutamine) jääks ära. Reaktsioon igale sümbolile peab järgnema kohe, mitte alles pärast järgmise sümboli sisselugemist.

- Kirjutada programm `Poora`, mis küsib loomulikus keeles kasutajalt sisendit, loeb klaviatuurilt rea (samal ajal kuvades selle ka ekraanile) ja toob rea lõppedes ekraanile selle rea sümbolid vastupidises järjekorras. Programm peab töötama tsüklis, kuni sisestatakse tühi rida.

- Kirjutada funktsioon, mis võtab argumendiks listi ja annab väärtuseks tegevuse, mis tsüklis ootab kasutajalt klahvivajutust: kui see on *enter*, kirjutab ekraanile listi järgmise elemendi; kui see on *escape*, lõpetab; kui see on midagi muud, ei reageeri kuidagi. Kui list on läbi, lõpetab programm kohe töö.

  – Realiseerida selle funktsiooni abil programm `Listid1`, mis kirjutab kasutaja *enter*-vajutuste peale koodide järjekorras kõik sümbolid (mitte vahetades iga sümboli järel rida).

  – Realiseerida selle funktsiooni abil programm `Listid2`, mis kirjutab kasutaja *enter*-vajutuste peale suuruse järjekorras naturaalarvude ruudud, igaühe eraldi reale.

Lisaks senivaadelduile on Haskellis olemas omapärane tegevus `getContents :: IO String`, mis seisneb kogu standardsisendi edastamises ühe sümbolite listina. List väärtustatakse seejuures laisalt, parajasti niipalju, kui tegevusele `getContents` järgnevad tegevused seda vajavad.

Selle tegevuse abil on prelüüdis defineeritud funktsioon `interact`:

```
interact f
  = getcontents >>= putStr . f .
```

Funktsiooni `interact` argumendiks on funktsioon, mis näitab, kuidas arvutada standardväljundisse saadetav string standardsisendist loetud stringi kaudu.

Niisuguse funktsiooni olemasolu teeb võimalikuks osade sisend-väljundit kasutavate programmide kirjutamise elegantsel viisil, esitades programmi kujul

```
main = interact f.
```

Kuna f `:: String -> String`, siis f sisend-väljundiga ei tegele. Ainus formaalselt sisend-väljundiga seotud programmiosa on seesama `interact`.

Kui `interact` argumentfunktsioon f vajab väärtuse leidmiseks kogu oma argumentlisti täies pikkuses, ei saa programm `interact` f muidugi normaalselt tööd lõpetada. Näiteks andes Hugsi käsurealt

```
interact reverse,
```

ootab süsteem järgnevalt kasutajasisendit lõpmatuseni ilma nähtava reaktsioonita, sest esimese sümboli standardväljundisse kirjutamiseks peaks see olema standardsisendis viimane, mis standardsisendit ümbersuunamata on võimatu. Samas

```
interact (const "Hei!")
```

kirjutab kohe ekraanile `Hei!`, ilma et oleks kasutajalt ühtki sümbolit oodanud, sest funktsioon `const "Hei!"` ei vaja väärtuse leidmiseks argumentstringi. Vahepealne variant on näiteks

```
interact (map toUpper),
```

mis kirjutab kõik kasutaja sisestatud sümbolid reaalajas ekraanile, kuid väiketähtede asemel paneb vastavad suurtähed. See programm küll normaalselt tööd ei lõpeta, kuid suudab pidevalt vastata kasutaja tegevusele.

- Katsetada Hugsi käsurealt tegevusi `interact` f erinevate funktsioonidega f. Üks neist olgu selline, mille puhul programm kirjutab kasutaja sisestatud tähed ekraanile, jättes muud sümbolid vahele.
- Kirjutada eelmise ülesandebloki ülesanded uuesti, nii et iga ülesande lahenduseks oleks programm kujul

  ```
  main = interact f.
  ```

Kuna `interact` seostab ainult standardväljundit standardsisendiga, siis on kujul `interact` f võimalik esitada ainult tegevusi, mis loevad ainult standardsisendit ja kirjutavad ainult standardväljundisse. Erinevate failidega ega käsureaargumentidega opereerimist ei saa selliselt programmeerida. Samuti pole võimalik selliselt teha erinditöötlust.

Erinditega mängimiseks on Haskellis järgmised funktsioonid.

1. Funktsioon `ioError :: IOError -> IO` a argumendil e **seab** erindi e. See tähendab, et `ioError` e väärtuseks on erindi e heitmine, mingit muud tegevust ei toimu.

   Seda funktsiooni võib kasutada ükskõik millist tüüpi väärtusi edastava tegevuse kohal. Seda näitab tema polümorfne tüüp, kus noole paremal pool on `IO` a vasakul pool määramata tüübiga a. Selles mõttes on `ioError` analoogne funktsiooniga `error`, mida samuti võib kasutada ükskõik millist tüüpi väärtuse kohal.

2. Funktsiooni `userError :: String -> IOError` väärtuseks argumendil s on erind, mis programmi poolt heites põhjustab programmi töö lõpetamise veateatega s. Selle funktsiooniga saab kasutaja niisiis erindeid ise koostada.

   Pangem tähele, et `userError` ei tegele erindiseadega. Erind on tema normaalseks väärtuseks, `userError` ei tegele sisend-väljundiga üldse. Erindi seadmiseks tuleb kasutada eelmises punktis kirjeldatud funktsiooni `ioError`. Niisiis seadmaks erindit, mis programmi poolt heites põhjustab veateate `Minu erind`, tuleb kirjutada

   ```
   ioError (userError "Minu erind").
   ```

3. Operatsioon `catch :: IO` a `-> (IOError -> IO` a`) -> IO` a on põhimõttelt sarnane operaatoriga >>=. Kui >>= tekitab tegevuste järgnevuse juhul, kui esimene tegevus lõpetab normaalselt, siis `catch`, vastupidi, teeb seda juhul, kui esimene tegevus heidab erindi. Kui tegevus a heidab erindi e, siis tegevus `catch` a h koosneb tegevusest a, millele vahetult järgneb tegevus h e; see tähendab muuhulgas, et `catch` a h edastab või heidab sama väärtuse, mille vastavalt edastab või heidab h e. Kui aga tegevus a edastab väärtuse x, siis `catch` a h tegevus on sama mis a-l; see tähendab, et funktsioonist h sel juhul midagi ei sõltu ning `catch` a h edastab sama väärtuse x.

   Selle funktsiooniga on võimalik **püüda** erindeid; tema teine argument kujutab endast **erinditöötlejat** (ingl exception handler), mis tekkinud erindist sõltuvalt annab välja alternatiivtegevuse. Tuleb tähele panna, et erinditöötleja väärtustüübi tegevused peavad edastama sama tüüpi väärtusi kui põhitegevus. Kui see tingimus pole täidetud, tekib tüübiviga.

Moodulis $\mathrm{Prelude}$ on ainult kõige elementaarsemad failioperatsioonid. Keerulisemad on koondatud moodulitesse $\mathrm{System.IO}$ ja $\mathrm{System.Directory}$. Käsureaargumentide kasutamiseks tuleb sisse lugeda moodul $\mathrm{System.Environment}$ või System.

- Kirjutada programm, mis võtab käsurealt kaks failinime ja kirjutab esimese faili need read, milles esineb täht `a`, teise faili. Kui käsureaargumente on rohkem või vähem kui 2, lõpetab programm kohe töö vastava teatega ekraanil. Kui faili lugemine või kirjutamine ebaõnnestus, lõpetab programm samuti kohe töö vastavasisulise teatega ekraanil. Püüda tekkinud viga võimalikult täpselt diagnoosida.