Haskelli interpretaator Hugsi kasutamine

- Käivitada Hugs, andes käsurealt käsu `hugs`.
- Teises aknas visata silm peale Hugsi standardteekidele.
- Teha endale selgeks Hugsi põhikäsud

  `:q` väljumine,

  `:?` Hugsi käskude nimekiri,

  `:s` eelistuste ja parameetrite nimekiri ja muutmine,

  `:e` tekstiredaktori avamine,

  `:l` mooduli sisselugemine,

  `:r` viimase sisselugemiskäsu kordamine,

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

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

  Sealhulgas `-Eemacs` seab tekstiredakoriks Emacsi ja `-P`... failiotsimistee.
- Lasta Hugsil väärtustada lihtsaid Haskell-avaldisi.
- Kirjutada ise programm, lugeda Hugsiga sisse ja töötada sellega. Esimesteks programmideks võiks olla faktoriaali arvutamine erinevatel viisidel.

Funktsionaalse paradigma eelis: kõrgemat järku funktsioonid

- Lisada faktoriaalimoodulile funktsioon `supFact`, mis mittenegatiivse täisarvu n korral tagastab väärtuse $n^{(n-1)^{\cdots 1}}$ (täisarvude astendamise märk Haskellis on `^`).
- Lisada veel ka funktsioon `subFact`, mis mittenegatiivse täisarvu n korral tagastab väärtuse $1 + 2 + \ldots + n$.
- Teha uus fail ja kirjutada seal kõrgemat järku funktsioon `kõrgFact`, mis, sõltuvalt lisaargumentidest (operatsioon ja algväärtus), teeb nii üht, teist kui kolmandat. Defineerida `fact`, `supFact` ja `subFact` erijuhtudena funktsioonist `kõrgFact`:

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

  Veenduda konkreetsete avaldiste väärtustamise teel, et funktsioon tõesti teeb seda, mis nõutud.
- Kirjutada kõrgemat järku funktsioon astendamiselaadsete operatsioonide (korrutamine, astendamine, hüperastendamine, ...; ühesõnaga, operatsioonid, kus üks argument ütleb, mitu korda tuleb teist argumenti iseendaga läbi “opereerida”) sooritamiseks. Kontrollida funktsiooni tööd samuti kui eelmises ülesandes.
- Arvutada eelmises ülesandes defineeritud astendamise abil avaldise

  $$\underbrace{\sin(\ldots(\sin}_{20} 100)\ldots)$$

  väärtus. Kasutada eeldefineeritud funktsioone

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

Mõned eritehnikad

- *Akumulaatorid* — rekursiivse funktsiooni argumendid, milles väärtustamise käigus asuvad arvutuste vahetulemused.
  - Kirjutada funktsioonide `kõrgFact` ja `kõrgAste` lokaalne funktsioon ümber akumulaatoriga, milles asuvad korrutamiste vahetulemused.
- *Laisk vs agar* väärtustamine: akumulaatoreid tahame tihti jooksvalt väärtustada.
  - Modifitseerida funktsioonide `kõrgFact` ja `kõrgAste` definitsioone nii, et akumulaatoris toimub jooksev väärtustamine.
  - Kirjutada funktsioon `fib`, mis naturaalarvulisel argumendil n annab väärtuseks Fibonacci jada liikme F_n.
- "*Jaga ja valitse*" tehnika.
  - Kirjutada funktsiooni `kõrgAste` uus variant, mis kasutades "jaga ja valitse" tehnikat annab logaritmilise ajakeerukuse. Võib eeldada, et operatsioon on assotsiatiivne (st sulge võib paigutada meelevaldselt).

Listid

- Tutvuda loengumaterjali näidetega listidel töötavatest funktsioonidest.
- Kirjutada funktsioonid
  - `vahed`, mis tühjal listil lõpetab veaga, mittetühjal listil aga tagastab vahede listi, nt `vahed [2,6,5]` $\Rightarrow$ `[4,-1]`;
  - `takeLast`, mis töötab nagu `take`, aga võtab elemente listi lõpust;
  - `myZipWith`, mis töötab nagu `zipWith`, aga ei viska ülejäävat listisaba ära, vaid jätab selle tulemuslisti lõppu;
  - `allDiff`, mis argumentlisti kohta otsustab, kas selle elemendid on paarikaupa erinevad;
  - `allEq`, mis argumentlisti kohta otsustab, kas selle elemendid on kõik võrdsed;
  - `disj`, mis kahe argumentlisti kohta otsustab, kas nad on disjunktsed, st tagastab `True`, kui listid ei sisalda ühiseid elemente, ja `False`, kui sisaldavad.
- Kuivõrd on kujul `foldr` f e väljendatavad
  - `head`, `tail`, `null`, `filter` p, `takeWhile` p, `dropWhile` p?
  - Teie poolt praktikumis kirjutatud funktsioonid?

  Väljendada, mis võimalik.
- Kuivõrd töötavad lõpmatutel listidel
  - loengumaterjalis toodud standardfunktsioonid?
  - `takeWhile` ja `dropWhile`?
  - Teie poolt praktikumis kirjutatud funktsioonid?
  - koduülesannetele antud lahendused?

  Kirjutada oma funktsioonid nii, et töötaksid lõpmatutel listidel võimalikult hästi. (See võib tähendada `foldr`-st loobumist.)

Sisend-väljund

- Moodulis `Prelude` on ainult kõige elementaarsemad failioperatsioonid. Keerulisemad on koondatud moodulisse `IO`. Käsureaargumentide kasutamiseks tuleb sisse lugeda moodul `System`.

- Et programm oleks käivitatav otse *shell*i käsurealt, tuleb failile anda käivitusõigus ning panna ta esimeseks reaks `#!/usr/local/bin/runhugs`.

- On võimalik kaks programmeerimisstiili.

  – Kasutada `do`-süntaksit ning funktsioone `>>=`, `>>` jms.

  – Esitada programm kujul

  $$\texttt{main} = \texttt{interact}\ \ f,$$

  kus f :: `String -> String` (kuna f tüüp ei maini `IO`-monaadi, on f tavaline funktsioon, mille definitsioon kusagil sisend-väljundiga ei tegele; seega ainus `IO`-monaadi kasutav koht koodis ongi seesama `interact`).

  Teises stiilis on võimalik opereerida ainult standardsisendi ja -väljundiga. Näiteks käsureaargumentidega tegelemist saab programmeerida ainult `IO`-monaadi käskudega. Ka erindeid ei saa tekitada ega püüda väljaspool `IO`-monaadi.

- Lahendada mõlemas stiilis järgmised ülesanded.

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

  – Kirjutada programm, 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 programm, mis ootab klaviatuurilt sümboleid, kuid ekraanile toob hoopis sümbolite koodid (sümboleid mitte), igaühe eraldi reale. Programm lõpetab töö, kui

  a) sisestatakse sama sümbolit kaks korda järjest.
  b) sisestatakse sümbol, mida on programmi töö jooksul juba sisestatud. Siin võiks lõpetamisel anda ka 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, 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 programmi, mis 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, mis kirjutab kasutaja *enter*-vajutuste peale koodide järjekorras kõik sümbolid (mitte vahetades iga sümboli järel rida).

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

- Lahendada järgmised ülesanded.

  – 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.