Automaatit ja kieliopit 2015 harjoitustyö
Janne Kauppinen

Tämän teksti on automaatit ja kieliopit 2015 kurssin harjoitustyön kirjallinen osio. Tässä tekstissä esitellään kurssia varten tehtyä tietokoneohjelmaa, ja erityisesti käydään läpi ne ohjelman piirteet, jotka liittyvät kurssin aihealueeseen.

Ohjelma on Haskell-ohjelmointikielellä toteutettu yksinkertainen sovellus, joka pyytää käyttäjältä aritmeettisia lausekkeita ja tulostaa joko virheilmoituksen tai lausekkeen numeerisen arvon. Sovellus kelpuuttaa syötteenä merkit "0123456789()+-*/eq". Q:ta painamalla ohjelman suoritus päättyy. Ohjelma tulkitsee numeroita liukulukunumeroina, joten esimerkiksi 1.53e-5, 13e3 ja -0.1234 ovat kelvollisia syötteitä. Sovellukselle syötettävä lauseke voisi olla esimerkiksi seuraavanlainen: (1+5)/(6-2)+33-11*4. Tällainen syöte antaa vataukseksi -9.5. Välilyönnit eivät ole sallittuja. Ohjelman pitäisi kääntyä ghc kääntäjällä esimerkiksi seuraavalla komennolla: ghc -o laskin parser.hs.

Ohjelma koostuu käytännössä katsoen kahdesta eri osa-alueesta: ensimmäinen osa-alue on merkkijonon jäsentäminen ja jäsennyspuun rakentaminen, ja toinen osa-alue on jäsennyspuun evaluointi. Käsitellään ensimmäiseksi läpi jäsennyspuolta.

Ohjelmassa käytettävä kielioppi BNF-muodossa:

```
<expression> ::= <constant>
               | <expression> + <expression>
               | <expression> - <expression>
               | <expression> * <expression>
               | <expression> / <expression>
               | (<expression>)
```

Haskelissa se on kirjoitettu seuraavaan muotoon, joka on jo sellaisenaan hyvin lähellä BNF-notaatioa:

```
data Expr a b = C a
              | b :+ b
              | b :- b
              | b :* b
              | b :/ b
              | Par b
              deriving (Functor, Show)
```

Ohjelma ottaa syötteenä merkkijonon, ja pyrkii jäsentämään saamansa merkkijonon kieliopinmukaiseksi rakenteeksi. Ohjelmassa jäsennin on koottu useasta primitiivijäsentimestä. Ohjelmassa käytettävät jäsentimet ovat muotoa:

```
newtype Parser a = P (String -> Maybe (a,String)).
```

Jokainen jäsennin pitää sisällään funktion, joka ottaa syötteenä merkkijonon, joka Haskelin tapauksessa on lista Char tyyppisiä arvoja, ja lista käyttäytyy käytännössä katsoen kuten pino. Jäsentimen funktio ottaa siis merkkijonon ja palauttaa paluuarvonaan Maybe (a,String):in. Jäsennin voi siis epäonnistua jäsentämisessä, jolloin lopputulos on Nothing, tai onnistuessaan se palauttaa Just (x,y), missä x on tyyppia a oleva jäsennetty arvo ja y on loput merkkijonosta, jota ei olla vielä käyty läpi. Jäsentäessään funktio siis kuluttaa pinoa ottamalla siitä aina päältä merkkejä, ja jos jäsennys onnistuu, niin loput merkkijonosta asetetaan y:h seuraavia jäsentimiä varten.

Parser on Functor-tyyppiluokan ilmentymä, eli jäsennin toteuttaa fmap-funktion. Fmap-funtion avulla voidaan luoda uusia jäsentimiä soveltamalla binääristä funktiota toisena ar-

gumenttinä annettun jäsentimen funktion paluuarvoon , erityisesti parin ensimmäiseen arvoon, joka on siis jäsenetty arvo. Fmap-funktio siis muodostaa uudenlaisen jäsentimen binäärisestä funktiosta ja annetusta jäsentimestä, kunhan binäärifunktion ensimmäisen parametrin tyyppi ja jäsentimen tyyppi a ovat samat. Fmap siis lisää uuteen jäsentimeen binäärifunktion toiminnalisuuden edellisen jäsentimen lisäksi. Jäsentimen täytyy olla funktori myös siksi, että se voisi olla Applicative-tyyppiluokan ilmentymä.

```
instance Functor Parser where
  fmap f (P fx) = P (fmap (\(x,s) -> (f x, s)) . fx)
```

Parser on myös Applicative-tyyppiluokan ilmentymä, joka puolestaan mahdollistaa jäsentimien ketjuttamisen ja tässä sovelluksessa sitä käytetään myös jäsennyspuun rakentamiseen, sillä Haskelissa rakentimen ovat funktioita, joten Applicativen käyttö on siten luontevaa. Jos jokin Applicativen ketjutetusta jäsentimestä epäonnistuu, tällöin koko jäsentimien ketju epäonnistuu. Parser on myös Monoid-tyyppiluokan ilmentymä, mikä puolestaan mahdollistaa jäsentimien vaihtoehtoistamisen. Mappend-funktio toimii tässä siten, että se ottaa argumentteina kaksi jäsennintä, ja muodostaa uuden jäsentimen. Tämä uusi jäsennin yhdistää argumentteinä saatujen jäsentimet siten, että se luo jäsentimen, jonka funktio testaa ensin ensimmäisenä parametrina saatua jäsennintä annetulla merkkijonolla, ja jos se epäonnistuu, niin se se kokeilee suorittaa seuraavaa jäsennintä, mutta jos ensimmäinen jäsennin onnistuu, se palauttaa ko. jäsentimen lopputuloksen. Jäsentimien vaihtoehtoistaimen menee siis vasemmalta oikealle, ja jos jokin jäsentimistä onnistuu, niin sen lopputulos jää voimaan.

Tarkasellaan seuraavaksi kielioppia. Tässä onjelmassa on käytetty PEG:iä (parsing expression grammar). PEG:in kielioppi näyttää samankaltaiselta kuin CFG-kieliopit, mutta tulkinta on kuitenkin erilainen. PEG perustuu siihen, että eri välikesymboleille määritellään niin sanottu prioriteetti järjestys, joka määrittelee sen missä järjestyksessä välikesymboleita kokeillaan. Tämä antaa ohjelmoijalle hyvän kontrollin esimerkiksi presedenssien määrittelyyn. Lisäksi PEG on deterministinen, eli sen luoma jäsennyspuu on yksiselitteinen, mikä on esimerkiksi artimeettisten lausekkeiden jäsentämisen kannalta hyvä asia. Lisäksi PEG mahdollistää tietyssä mielessä rajattoman ennakoinnin ja peruuttamisen, mikä helpottaa joitakin asioita kuten esimerkiksi roikkuvan else:n ongelman tai * ja ** symbolien erottamisen toisistaan, mutta toisaalta se voi aiheuttaa jäsennysprosessiin eksponentiaalisen suoritusajan, kuten tässäkin ohjelmassa käy. Seuraavaksi käsitellään kielioppia, jossa on asetettu välikemerkeille prioriteettijärjestys.

```
expr2 = e
    where e = add <> sub <> t
          t = mult <> div <> p
          p = par e <> cons
          cons = fmap Fx $ (pure C) <*> double
          add = fmap Fx $ pure (:+) <*> (t <* string "+") <*> e
          sub = fmap Fx $ pure (:-) <*> (t <* string "-") <*> e
          mult = fmap Fx $ pure (:*) <*> (p <* string "*") <*> t
          div = fmap Fx $ pure (:/) <*> (p <* string "/") <*> t
```

```
E = T + E | T - E | T
T = P * T | P / T | P
P = c | (E)
```

Edellä on siis esitetty kielioppia sekä ohjelman koodina että välike- ja päätemerkkimuodossa. (<>)-funktiolla eli mappend-funktiolla on vaihtoehtoistettu eri jäsentimiä, eli tässä tapauksessa ensin kokeillaan (T+E):tä, ja jos se epäonnistuu, niin kokeillaan (T-E):tä. Loogisesti katsottuna epäonnistumistilanteessa peruutetaan aina takaisin ja kokeillaan seuraavaa vaihtoehtoa. Tällä tavalla jäsennyspuusta tulee aina yksikäsitteinen, ja kokeilujärjestys

määrittää jäsennyspuun rakenteen. Prioriteettijärjestyksen muuttaminen saattaa kuitenkin rikkoa kieliopin siten, että se kuvaa vääränlaisia kieliä verrattuna alkuperäiseen kielioppiin. Ohjelmoijan on oltava tarkka prioriteetijärjestyksestä.

Kustakin välikesymbolista on tehty oma funktionsa, ja kun syöte on käyty läpi, niin expr2-funktio tuottaa lopullisen jäsentimen. Applicativeä on käytetty välikemerkki-funktioissa, ja itse jäsennyspuun solmun rakennin annetaan pure-funktiolle, jolloin se saadaan jäsentimen sisälle ja voidaan soveltaa Applicativen ominaisuuksia (<*>, <*, *>). Ex tuottaa siis lopullisen jäsentimen, ja runParser-funktiolla voidaan suorittaa jäsennin annetulla merkkijonolla. Jäsennys etenee vasemmalta rekursiivisesti, ja kieliopin pitäisi tuottaa aritmeettisia lausekkeita oikein. Kieliopissa ei ole välitöntä eikä epäsuoraa vasenta rekursiota. Tässä vaiheessa on myös tehty fmap:ia käyttäen fiksaus Fx-rakentimella. Tämän olisi ehkä voinnut tehdä myös myöhemminkin, mikä olisi helpottanut debuggausta.

Ylläoleva kielioppi ei tosin ole hyvä ratkaisu, sillä sisäkkäisten sulkeiden tapauksessa päädytään eksponentiaalisen ajan ongelmaan. Tämä johtuu käytännössä katsoen siitä, että sulkeet tarkistetaan aina lopuksi, ja jos sulkeita on paljon, niin tällöin lasketaan turhaan samat välitulokset kullekin syöttömerkille. Tässä tapauksessa siis joudutaan kokeilemaan yhä uudestaan turhaan kaikki mahdolliset vaihtoehdot ennen sulkeita, vaikka ne olisi jo aiemmin laskettu. Tämä toteutustapa ei siis muista edellisten vaihtoehtojen tuloksia, vaan samat laskennat suoritetaan yhä uudestaan. Vaikka ohjelma antaakin oikean lopputuloiksen, niin esimerkiksi seuraavan syötteen jäsennys kestää useita minuutteja: (((((((7))))))). Sulkeiden siirtäminen ihan alkuun korjaa edellämainitun sulkuongelman, mutta rikkoo kaiken muun kieliopin. Myös monia muita eri vaihtoehtoja käytiin läpi ohjelman suunnittelussa, mutta näyttäisi siltä, että hyvän PEG-kieliopin määrittäminen siten, ettei päädyttäisi jonkinlaiseen eksponentiaaliseen ongelmaa näyttäisi olevan melko pottumaista.

Eräs mahdollinen ratkaisu eksponentiaalisen ajan pulmaan voisi olla niin sanotut packrat-jäsentimet. PEG-kielioppi pysyisi samana, mutta jokaista syötemerkkiä kohden luotaisiin tietorakenne, johon tallennettaisiin laskettu välitulos täsmälleen yhden kerran. Tällä tavalla pystyttäisii välttymään välitulosten laskemisen useaan kertaan. Toisaalta tällöin välitulosten muodostaman "taulukon"koko on n*(m+1), missä n on välikemerkkien lukumäärä ja m on merkkijonon pituus. Toisinsanoen packrat-jäsentimien tarvitseman muistin määrä kasvaa merkittävästi, jos annettu merkkijono on iso. Toisaalta nykytietokoneilla on melko hyvin muistikapasiteettia. Packrat-jäsentimien sanotaan toimivan lineaarisessa ajassa, jos jäsennin on ohjelmoitu hyvin. Packrat-jäsentimien haaste on kuitenkin se, että huonosti ohjelmoituna ne saattavat laskea turhia välituloksia. Tosin laiskassa ohjelmoitikielessä kuten Haskelissa on mahdollista rakentaa tietorakenne niin, että lasketaan vain ne välitulokset jotka on pakko laskea, ja jätetään laskematta epärelevantit tulokset.

Toinen varteenotettava tapa välttyä eksponenttiaalisen ajan ongelmalta olisi ollut CFG:t. Tämän ohjelman kielioppi on sen verran yksinkertainen, että CFG olisi ollut helppo toteuttaa. Kieliopille olisi siinä tapauksessa pitänyt tehdä esimerkiksi väsen tekjöinti ja mahdollisesti vasemman rekurssion poisto. Myös esimerkiksi LL(1)-kieleksi määrittely olisi varmaankin ollut kohtalaisen helppo toteuttaa tälle kieliopille, jolloin suoritusaika olisi ollut nopea. Jos tässä ohjelmassa olisi luettu syötteenä esimerkiksi luonnollisia kieliä, tai jos haluttaisiin jostain syystä saada laskettua jäsennyspuulle eri vaihtoehtoja, olisi CFG ollut selkeästi parempi vaihtoehto kuin PEG. PEG soveltuu hyvin koneiden muodostamille kielille, mutta jos halutaa epädeterminististä ja järjestysriippumatonta toteutustapaa, niin tällöin CFG-kielet ovat selkeästi parempi vaihtoehto. PEG puolestaan antaa ohjelmoijalle hyvät mahdollisuudet tuottaa kielioppeja, jotka lukevat syötettä pitkälle ja kokeilee erilaisia vaihtoehtoja, mutta haasteena tulee eksponettiaalisen ajan onglemat, mahdollisesti muistikäyttö ongelmat ja vaikeudet saada kielioppi määriteltyä niin, ettei se tuota vääränlaisia kieliä.

Ohjelmassa on myös määritelty CFG-kielioppi double:n jäsentämistä varten. Kielioppi on seuraavanlainen:

```
S -> MA | ZA | ZB | d
A -> ZA | ZB | d
B -> XC | YZ
C -> PF | MF | ZF | d
D -> ZD | ZE | f
E -> XC
F -> ZF | d
P -> +
M -> -
X -> e
Y -> .
Z -> d
```

Itse ohjelmakoodi on double-funktion kohdalla. Tämä ratkaisutapa juontaa juurensa lähinnä akateemisesta mielenkiinnosta eikä sen vuoksi, että se olisi välttämättä hyvä ratkaisu tässä tapauksessa. Kielioppi on nyt muokattu chomskyn normaali-muotoon. Näin aikaansaatu jäsennyspuun pitäisi olla ainakin teoriassa binäärinen jäsennyspuu. Tässä on lähinnä testattu, että saadaanko kielioppi kirjoitettua kutakuinkin sellaisenaan ohjelmakoodiksi. Koodissa on pieni testiohjelma joillekin testimerkkijonoille, jotka jäsentyvät Haskelin tunnistamaksi doubleksi. Tämä kielioppi on ensin piirretty ratakiskoksi, josta sitten ajattelutyöntuloksena saatettu kontekstittomaksi kieliopiksi, jonka jälkeen on poistettu yksikköproduktiot ja tämäm jälkeen saatettu chomskyn normaalimuotoon, tavoitteena binäärinen jäsennyspuu. Chomskyn normaalimuoto ei siis ole olennainen tässä tapauksessa, sillä liukulukujen jäsentämisessä ei todennäköisesti päädytä kovinkaan syviin jäsennyspuihin.

Seuraavaksi käsitellään jäsennyspuun tyyppiä ja evaluointia. Määritellään tyyppirakennin Fix.

```
Fix f = Fx (f (Fix f))
```

Kyseessä on tyypin "kiintopiste". Toisin sanoen Fix:ä käyttämällä jäsennyspuun tyyppi säilyy samana riippumatta jäsennyspuun solmujen lukumäärästä. Jäsennyspuun rakennusvaiheessa jokainen primitiivi annetaan Fx:n rakentajalle. Tyyppi (Expr a b) on funktori, joten jäsennys puu voidaan käydä läpi rekursiivisesti cata-funktiota ja fmap-funktioa käyttäen. Tämä ominaisuus antaa mahdollisuuden foldata jäsennyspuu läpi monimutkaisellakin algebralla. Määritellään seuraavaksi Algebra, cata ja unFx.

```
type Algebra f a = Functor => f a -> a

cata :: Functor f => (f a -> a) -> Fix f -> a
cata algebra = algebra . fmap (cata algebra) . unFx

unFx :: Fix f -> (f (Fix f))
unFx (Fx x) = x
```

Algebra määrittelee säännöt sille, miten a:t operoidaan. Samassa sovelluksessa voisi siis olla määritelty useita eri algebroja. Tässä ohjelmassa on määritelty tyyppiluokka Eval, jonka ilmentymien täytyy toteuttaa funktio evalAlgebra. Sovelluksessa on määritelty evalAlgebra RealFrag-tyyppisille arvoille, jonka Double:kin toteuttaa. EvalAlgebra toimii siis monelle muullekin numeeriselle tyypille. Ohjelmakoodissa on myös algS-funktio, joka on algebra Stringeille. Tosin vain + ja * operaatiot on määritelty. Tässä ohjelmassa ei käytetä Stringejä, mutta ne ovat mukana ohjelmakoodissa demonstraation vuoksi.

Cata on rekursiivinen funktio, joka ottaa argumenttinä algebran ja jäsennyspuun. Cata toimii käytännössä siten, että se purkaa ensimmäisen Fx:n, joka on siis alussa jäsennyspuun juuri. Tämän jälkeen päästään fmap:ia käyttämällä soveltamaan cata funktiota uudestaan annetulla algebralla seuraavaan solmuun. Tällä tavoin käydään koko jäsennyspuu läpi, kunnes tilanne on se, että evalAlgebralla on käytässään oikeita arvoja, joihin soveltaa evalAlgebraa. EvalAlgebra-funktiolle voitaisiin määritellä myös monimutkaisempiakin tehtäviä kuten esimerkiksi se, että lasketaan kuinka monta yhteenlaskua lauseke sisälsi. Fix:n haasteet todennäköisesti tulevat paremmin ilmi, jos kielioppia laajennettaisiin. Tällöin jouduttaisiin todennäköisesti käyttämään GADT:ja (generalized algebraic datatypes). Itselläni oli aikomus toteuttaa hieman monipuolisempi kieli GADT:ien ja Fix:n avulla, mutta käytännössä jäsentäminen vei niin paljon aikaa, ettei edellämainittuihin asoihin ollut aikaa. Itse jäsennyspuun solmujen fiksaus määriteltiin jäsentimen luomisen yhteydessä (add, sub, mul, div, cons). Lopputuloksena saatiin Parser DoubleExpr, jossa DoubleExpr on tyyppiä Fix (Expr Double), ja Double on laskennan lopputuloksen tyyppi, ja tämä tyyppi määrittelee käytännössä myös sen, mitä algebraa käytetään.

Itse evaluointi tapahtuu kahdessa eri vaiheessa. Ensin suoritetaan evaluate-funktio. Funktiolle annetaan jäsennin ja merkkijono, ja jos jäsennys epäonnistuu, niin se palauttaa Nothing. Jos jäsennys onnistuu, mutta syötettä ei ole luettu loppuun, niin tämäkin tulkitaan epäonnistumiseksi. Jos jäsennys onnistuu ja syötemerkkijono on käytetty loppuun, niin suoritetaan varsinainen evaluointi cata-funktiolla, evalAlgebralla ja fiksatulla konkreettisella jäsennyspuulla. Cata-funktio on siis evaluoinnin jälkimmäinen ja varsinainen evaluointivaihe.

Käsitellään seuraavaksi ratkeavuuskysymyksiä. Tässä ohjelmassa ei ole loputtomia rekursioita, ja vaikka joidenkin syötteiden aikavaatimus onkin exponentiaalinen, niin ohjelman antaa lopulta jonkinlainen vastauksen. Jos tämän ohjelman kielioppi ei pysty määrittelemään syötettä, niin se antaa virheilmoituksen. Muussa tapauksessa ohjelma laskee Haskelin omilla tietotyypeillä ja aritmeettisilla funktiolla vastauksen siinä rajoissa, minkä Haskel mahdollistaa. Tietyn rajan jälkeen Haskell antaa vastaukseksi Infinity:n tai -Intinity:n. Haskell-ohjelmointikieli asettaa siis tietynlaiset laskennan rajat, mutta kielioppi pystyy tunnistamaan syötteiden syntaksin hyvin. Tämän ohjelman toiminta onkin hyvin pitkälle sitä, että tutkitaan kuuluuko annettu syöte tiettyyn PEG:n määrittelemään kieleen. Tämänkaltaiset päätösongelmat ovat luonteeltaan ratkeavia.

Loppuyhteenvetona voisin sanoa sen, että onjelmointityö oli melko opettavainen prosessi. Ehkä yksi suurimpia asioita jonka ohjelmointityö opetti oli se, että kieliopin valinta ei ole aivan yhdentekevää varsinkaan silloin, kun sen joutuu ohjelmoimaan alusta alkaen itse. PEG:n hyödyt ja haasteet tulivat ohjelmointityössä hyvin esille. Aikaa kului ehkä liikaa Fix:n ymmärtämiseen ja Haskelin tyyppijärjestelmän kanssa tappeluun, mikä ei oikeastaan kuulunut tämän kurssin aihepiiriin. Aikaa kului myös turhan paljon sulkuongelman pohtimiseen. Ajan olisi voinut käyttää ehkä viisaaminkin. Toisaalta tämä onjelmointityö antoi jonkinlaisen näkemyksen siitä, miten tosielämän jäsentimet suunnilleen toimivat.