

AS-0.3301 - projektidokumentti

Aihe - *Symbolic calculator*

Petteri Hyvärinen, 67316W, ohyvarin@cc.hut.fi
Ian Tuomi, 79304V, ituomi@cc.hut.fi
Toni Rossi, 78731S, torossi@cc.hut.fi

16. joulukuuta 2010

Tiivistelmä

Tämä dokumentti kuvaa ryhmän `symbolic-1` toteutuksen symbolisesta laskimesta. Laskinta käytetään komentorivipohjaisella käyttöliittymällä, jolle annetaan syötteenä matemaattisia lausekkeita, jotka laskin pyrkii sieventämään.

1 Ohjeita ohjelman kääntämiseen ja suorittamiseen

1.1 Kääntäminen

Ohjelman kääntäminen tapahtuu projektin hakemistossa komentoriviä käyttäen seuraavalla tavalla:

```
1 kosh symbolic1 1048 % cd src
2 kosh src 1049 % make
```

Komento `make` kääntää ohjelman ja luo kansioon suoritettavan tiedoston `calculator`.

1.2 Ohjelman käyttö

Kääntämisen jälkeen `src`-alihakemistossa on suoritettava tiedosto `calculator`, jonka suorittaminen käynnistää ohjelman komentorivin:

```
1 kosh src 1057 % ./symbolic
2
3 Welcome to Symbolic-1!
4 Type "help" for help.
5
6 >>
```

Ohjelmaa käytetään komentorivin kautta. Käytössä ovat seuraavat komennot (listauksen saa komentorivin komennolla `commands`):

```
1 >> commands
2
3 Commands
```

```

4 -----
5
6   who          Displays list of defined functions and variables
7   clear        Clears defined functions and variables
8   simple       Toggles autosimplify (default: on)
9   prefix       Turns prefix mode on
10  infix        Turns infix mode on (default)
11  help          Displays help
12  commands     Displays this list
13  exit/bye      Ends program

```

1.3 Syntaksi

Sievennettävät lausekkeet syötetään komentoriville ja ohjelma sieventää syötteen automaattisesti, ellei automaattista sievennystä ole asetettu pois päältä komennolla `simple`. Lausekkeen voi halutessaan tallettaa funktioksi komentorivin muistiin, esimerkiksi seuraavilla tavoilla:

```

1   >> f[x,y] = x^2+y^3+x
2   >> h[m]=f[m^2,m^3+2] # sieventyy ((8+m+m^2)^m)^3+2
3   >> aleph = h[3]      # sieventyy ((8+3+3^2)^3)^3+2 -> 5.12e+11
4   >> aleph            # sieventyy 5.12e+11

```

Jos lauseketta ei tallenneta funktioksi, se tallentuu automaattisesti `ans`-nimiseksi funktioksi. Funktion `ans` arvo korvautuu uudella funktiolla siten, että vain viimeisin lauseke on muistissa. Jos lausekkeessa esiintyy muita symboleja kuin aikaisemmin tallennettujen funktioiden tunnisteita, ne tulkitaan muuttujiksi. Yhtäsuuruusmerkkiä täytyy edeltää syötteen aloittava nimi ja mahdolliset funktion muuttujamäärittelyt, esim. `f[x,y]`. Jos määrittelyitä ei ole tehty, tehdään ne implisiittisesti aakkosjärjestyksessä. (esim. `f[x,y] = x+y` vastaa määrittelyä `f = x + y`) Nimissä ovat sallittuja vain suuret ja pienet kirjaimet. Esim. "Function2" ei ole sallittu nimi mutta "FunctionTwo" olisi. Yhtäsuuruusmerkkiä täytyy seurata funktio. (esim. `munFunktio = x^2 + x^3 + sin x`). Myös funktion evaluointi tapahtuu hakasulkeiden avulla. Jos lauseke, joka sisältää muuttujia, tallennetaan funktioksi, saatu funktio voidaan evaluoida halutussa pisteessä kutsumalla funktiota parametrilistalla. Parametrien arvot voivat olla mitä tahansa laillisia syötteitä, esimerkiksi lukuja, symboleita tai funktioita, esimerkiksi `f[7]` toimii odotetulla tavalla ja `f[x^2]` korvaa `x`:n funktiolla `x^2`.

Ohjelma hyväksyy oikeanpuoleiselle funktiolle syötteeksi Lispistä tuttua prefix-notaatiota tai perinteistä infix-notaatiota. Alussa ohjelma tulkitsee syötteen infix-syötteeksi mutta kirjoittamalla `prefix` vaihtaa ohjelma syöteodotettaan. Vastaavasti `infix` vaihtaa takaisin "perinteiseen" notaatioon. Infix-notaatio muunnetaan prefix-notaatioksi käyttäen mukailtua Dijkstran "Shunting Yard"-algoritmia. Prefix-notaatiota ei käsitellä sen syvällisemmin, siitä vain poistetaan sulut.

2 Ohjelman rakenne

Käsitlemme kaikkia syötteitä matemaattisina funktioina, joten kaikki luokat periytyvät abstraktista luokasta `Function`. Matemaattisen syötteen tulkitseminen funktioina ja funktioiden yhdistelminä on teoreettisesti perusteltua ja sopii

hyvin olio-ohjelmoinnin periaatteiden kanssa lähtien liikkeelle yleisestä mallista, tarkentuen eri ilmiöihin alemmilla tasoilla. Algebralliset funktiot pitävät erikoistapauksina sisällään mm. vakiofunktiot $f(x) = c$, identiteettikuvauksen $f(x) = x$ ja lisäksi polynomit ja juurifunktiot. Myös yksinkertaiset aritmeettiset operaatiot voidaan nähdä funktioina, esimerkiksi $f(x) = x + c$, $g(x, y) = x*y$ jne.

Luokkarakenne on kuvattu UML-kaaviolla liitteessä A. Periaatteena on ollut jaotella operaatiot ensin pariteetin mukaan ja sen jälkeen periyttää eri laskutoimitusten toteutukset omiin luokkiinsa. Luokkarakenne on suunniteltu siten, että jokainen funktio-olio sisältää paitsi kuvaamansa operaation toteutuksen, myös osoittimen tai osoittimia operandeihinsa. Näin funktio-oliot yhdessä muodostavat puurakenteen, jonka solmuina ovat operaatiot ja lehtinä nollaariset operandit - muuttujat tai luvut. Ohjelma tulkitsee käyttäjän syötteen ja rakentaa lausekkeen mukaisen puun funktioista. Sievennykset sekä lausekkeen evaluointi tapahtuu rekursiivisesti käymällä läpi koko funktiopuu. Suurin osa toteutetuista operaatioista on binäärisiä, eli niillä on kaksi operandia. Toteuttamamme luokkarakenne antaa kuitenkin mahdollisuuden laajentaa toiminnallisuutta myös useampioperandisiin operaatioihin.

2.1 Funktio-luokat

Tässä osassa kuvaillaan **Function**-luokasta periytyvät luokat, jotka matemaattisen lausekkeen eri alkioita, kuten vakioita, muuttujia tai laskuoperaatioita. Tarkemmat kuvaukset luokkien toiminnasta löytyvät luvusta 3. Jokaisella laskuoperaatiolla vastaavalla luokalla on omat .hh ja .cc tiedostonsa, jotka löytyvät **src**-alihakemistosta. Tämän lisäksi hakemistosta löytyvät käyttöliittymän toteutus tiedostossa **main.cc** sekä syötteen tulkinna toteutus tiedostossa **parser.cc**.

2.1.1 Abstraktit yläluokat

Funktio-luokkahierarkian juuressa on luokka **Function**, joka kuvaa mielivaltaista matemaattisen lausekkeen alkioita. Kaikki operaatiot ja alkiot perivät tämän luokan. **Function**-luokalla on yksi private-jäsen, joka on **std::set<Variable>**-tyyppinen kokoelma. Tämä kokoelma kuvaa muuttujia, joita kyseisellä funktiolla on. **Function**-luokassa on määritelty useita virtuaalisia jäsenfunktioita, jotka näin ollen periytyvät luokkahierarkiassa. Näihin kuuluvat mm. **simplify()**, joka palauttaa uuden, sievennetyn funktiopuun, **evaluate(std::map<Variable, Function*>, Function*>)**, joka palauttaa uuden funktiopuun, jossa muuttujien arvot on määrätty parametrina saatavassa map:issa. Näiden funktioiden toimintaa kuvataan luvussa 3. Kaiken kaikkiaan luokalla on seuraavat jäsenfunktiot:

```

1 Function()
2 Function(Variable variable)
3 Function(std::set<Variable> variables)
4 Function(const Function& other)
5 virtual ~Function()
6 virtual Function* simplify() const
7 virtual Function* evaluate(std::map<Variable, Function*>
    parameters) const
8 virtual std::set<Variable> getVariables() const
9 virtual Function* clone() const
10 virtual std::string print() const

```

```

11 virtual Function* derive(const std::string) const
12 virtual int getFunctionType() const
13 virtual bool isFunctionType(int x) const
14 Function* simplifyTree() const

```

Edellä mainittujen `simplify()` ja `evaluate(...)`-funktioiden lisäksi jokaisella `Function`-luokan perivällä luokalla on siis konstruktoreiden lisäksi jäsenfunktiot `getVariables`, joka palauttaa funktion muuttujat, `clone()`, joka palauttaa kopion ko. funktiosta, `print()`, joka palauttaa funktion merkkijonoesityksen, `derive(const std::string)`, joka palauttaa funktion derivaatan parametrina saadun muuttujan mukaan. Funktioita `getFunctionType()` ja `isFunctionType(int)` käytetään funktio-olion tyyppin määrittämiseen esimerkiksi sievennettäessä funktiopuuta. Eri funktiotyypeille on määritelty tiedostossa `function.hh` kokonaislukuarvot siten, että esimerkiksi muuttuja-tyyppinen funktio-olio palauttaa `getFunctionType()`-funktiota kutsuttaessa kokonaisluvun 1.

`NaryOperation` on abstrakti luokka, joka kuvaa n-ariteettista operaatiota. Sillä on `protected`-jäsenenään `std::vector<Function*>`-tyyppinen, `_operands`-niminen kokoelma, joka sisältää luokan kuvaaman operaation operandeja. Operandeja voi tarkastella erikseen kutsumalla `getOperands()`-jäsenfunktiota, joka palauttaa kopion em. kokoelmasta.

`NullaryOperation`, `UnaryOperation` sekä `BinaryOperation` perivät `NaryOperation`-luokan, ja kuvaavat nimiensä mukaisesti operaatioita, joiden ariteetit ovat vastaavasti 0, 1 sekä 2. Nämä luokat säilyttävät operandejaan `NaryOperation`-luokalta perimässään `_operands`-vektorissa, jonka ne jokainen alustavat itselleen sopivan kokoiseksi. `BinaryOperation`-luokkaan on lisätty jäsenfunktiot `getRhs()`, `setRhs(Function*)`, `getLhs()`, `setLhs(Function*)`, helpottamaan operandien käsittelyä vektorin käsittelyn sijasta.

Edellä esitellyt luokat ovat kaikki määritelty tiedostossa `function.cc/hh`.

2.1.2 Vakiot

Vakiot ovat lukuarvoja lausekkeessa. Vakiolla on `private`-jäsenenään `Rational`-luokan olio. Näin ollen vakiot voivat olla mielivaltaisia murtolukuja. Vakioiden luokka on `Constant`, joka on määritelty tiedostoissa `constant.cc/hh`. Vakio-oliolla ei ole ainuttakaan operandia, joten se perii `NullaryOperation`-luokan.

2.1.3 Muuttujat

Muuttujat ovat symboleja, jotka kuvaavat muuttuvaa arvoa lausekkeessa. Lausekkeessa esiintyvä muuttuja voidaan funktiossa korvata millä tahansa arvolla. Muuttujien luokka on `Variable`, joka on määritelty tiedostoissa `variable.cc/hh`. Myöskään muuttujalla ei ole ainuttakaan operandia, joten sekin perii `NullaryOperation`-luokan. Muuttujilla on `private`-jäsenenään `std::string`-tyyppiset kentät `_symbol` sekä `_namespace`, joista ensimmäinen kertoo muuttujan symbolin funktiossa ja jälkimmäinen nimiavaruuden, jossa muuttuja on määritelty. Tällä järjestelyllä pyritään välttämään eri funktioissa määriteltyjen muuttujanimien törmäykset. Nimiavaruus määräytyy sen funktion mukaan, jossa muuttuja on määritelty.

2.1.4 Yhteenlasku, kertolasku, jakolasku sekä potenssi

Yhteenlasku, kertolasku, jakolasku sekä potenssi ovat binäärisiä operaatioita, joten niitä kuvaavat luokat `Sum`, `Multiplication`, `Division` sekä `Power` perivät luokan `BinaryOperation`. Luokat on toteutettu tiedostoissa `sum.cc/hh`, `multiplication.cc/hh`, `division.cc/hh` sekä `power.cc/hh`. Kaikissa operaatioissa operandit voivat olla mitä tahansa `Function`-luokan periviä olioita.

2.1.5 Negaatio

Negaatio on unaarinen operaatio, joka muuttaa operandinaan olevan funktion etumerkin. Luokka `Neg` on määritelty tiedostossa `neg.cc/hh` ja se perii luokan `UnaryOperation`.

2.2 Apuluokat

Tässä osassa kuvaillaan funktiorakenteen ulkopuoliset luokat, joiden tarkoituksena on helpottaa funktio-olioiden käsittelyä.

2.2.1 Rationaaliluvut

Rationaalilukuja kuvaa luokka `Rational`, joka on toteutettu tiedostoissa `rational.cc/hh`. Luokassa on toteutettu kaikki peruslaskutoimitukset rationaaliluvuille. Rationaalilukuja käytetään vakioiden yhteydessä.

2.2.2 Polynomit

Polynomien käsittelyä varten tehtiin omat luokkansa `Polynomial` ja `Term`, joiden toteutus on tiedostoissa `polynomial.cc/hh`. Luokka `Term` kuvaa polynomitermiä, ja luokka `Polynomial` kokonaista polynomia, joka koostuu vektorista `Term`-olioita. Luokka on erillinen funktiorakenteesta, sillä sen tarkoitus on olla apuluokkana sievennyksessä. `Polynomial`-olio voidaan luoda joko funktiopuusta, `Term`-vektorista tai ottamalla kopio toisesta polynomioliosta. Luokalla on myös parametrin konstruktori, joka luo tyhjän polynomin.

3 Tietorakenteet ja algoritmit

Kuten luvussa 2 todettiin, lausekkeen alkiot tulkitaan funktioiksi, jotka yhdessä muodostava puurakenteen.

3.1 Sievennys ja evaluointi

Funktioiden toteutukset ovat käytännössä jäsenfunktiossa `simplify()`, joka palauttaa yksinkertaisimman mahdollisen version lausekkeesta. Jos funktio ei löydä itsestään sievempää versiota, palauttaa se oman osoitteensa, jolloin onnistuneen sievennyksen onnistuminen on helppo todentaa.

Sievennettäessä kokonaista funktiopuuta kutsutaan funktiota `simplifyTree`, joka kutsuu puun pään `simplify`-funktioita yhä uudestaan kunnes se palauttaa oman osoitteensa. Jos laskutoimitus ei sievene välittömien lastensa suhteen, kutsutaan lasten vastaavia `simplify`-funktioita. Näin puu voidaan iteroida pohjia

myöten läpi. Huomionarvoista on, että puu palautuu `simplifyTree`-funktiolle välittömästi jokaisen onnistuneen sievennyksen jälkeen.

Funktiota evaluotaessa, muuttujat korvataan parametrien arvoilla (ks. 3.1.2), ja näin saatu puu jälleen sievennetään.

3.1.1 Vakiot

Yksinkertaisimmat funktio-oliot ovat vakioita, jotka ovat jo itsessään yksinkertaisimmassa muodossaan. Niinpä kutsuttaessa vakion `simplify()`-jäsenfunktiota, paluuarvona saadaan vakio itse. Samoin jäsenfunktio `evaluate(std::map<Variable, Function*>)` palauttaa yksinkertaisesti kopion itsestään.

3.1.2 Muuttujat

Muuttujien kanssa toimitaan samaan tyyliin, sillä erolla, että funktiota evaluotaessa lausekkeesta löytyvä muuttuja korvataan `evaluate(std::map<Variable, Function*>)`-funktiossa parametrina saatavasta map:ista etsittävällä arvolla. Funktiossa katsotaan, onko parametrina saadussa map:issa jäsentä, jonka key se itse olisi. Jos näin on, funktio palauttaa tätä key:tä vastaavan arvon, joka on `Function`-tyyppinen osoitin. Tämä osoitin vastaa komentorivillä annettua parametriä, esimerkiksi tapauksessa, jossa funktio `f[x] = x` evaluoidaan arvolla `x = a+2*a` – jolloin komentorivin syöte olisi `f[a + 2*a]` – korvattaisiin muuttuja `x` funktiolla `'+'`, jonka operandina olisivat edelleen `a` ja `2*a`. Tämän jälkeen näin saatu uusi funktiopuu voitaisiin sieventää ja saada tulos `3*a`.

3.1.3 Yhteenlasku, kertolasku, jakolasku sekä potenssi

Näiden operaatioiden sieventäminen tapahtuu niin ikään `simplify()`-jäsenfunktiossa. Kutsuttaessa kyseistä jäsenfunktiota, kukin funktio-olio luo operandeistaan sievennetyt versiot kutsumalla niiden omia `simplify()`-funktioita. Tämän jälkeen funktio-olio tarkastelee operandejaan ja selvittää niiden tyyppin, jonka jälkeen se etsii mahdollista sievempää muotoa.

Esimerkiksi tilanteessa, jossa sievennettävä lauseke on $1 + x + 2$, muodostetaan puu, jonka juuressa on `Sum`-tyyppinen funktio-olio operandeinaan vakio sekä toinen `Sum`-olio. Kun tätä lauseketta lähdetään sieventämään, juuren summafunktio kutsuu kummankin operandin `simplify()`-funktioita ja saa takaisin vakion 1 sekä summa-olion, sillä yhteenlasku $1 + x$ ei sievene. Tämän jälkeen algoritmi käy puuta läpi niin kauan kuin puussa on peräkkäisiä summia ja yrittää aina laskea yhteen kahta summalausekkeen jäsentä. Tässä tapauksessa yhteenlasku onnistuu kun $1 + 1$ palauttaa vakion 2. Tämän jälkeen puu ei enää sievene, joten tulos on $2 + x$. `simplify()`-funktio tekee myös sulkujen auki laskemista ja näin helpottaa omaa työtään. Esimerkiksi jos summa-oliolla on operandeinaan kaksi summa-oliota, sievennysalgoritmi muokkaa puuta siten, että sen oikeanpuoleinen operandi on jokin muuta kuin summa, käytännössä muuttaen lausekkeen $(a + b) + (c + d)$ muotoon $((a + b) + c) + d$.

Polynomien jakolaskussa käytetään jakokulma-algoritmia. [1, s. 9]

`simplify()` ja `evaluate(...)` palauttavat aina uuden funktio-olion ja näin ollen kutsuttaessa kyseisiä jäsenfunktioita funktiopuun juurelle, paluuarvona on kokonainen uusi funktiopuu.

4 Tiedossa olevat bugit

Funktion evaluoinnin tulisi toimia myös käyttöliittymään tallennetuilla funktioilla, esimerkiksi seuraava syöte:

```
1 f[x] = x^2
2 g[y] = y
3 f[g[2]] # should return 4
```

Tämä ei kuitenkaan onnistu, vaan käyttöliittymä antaa virheilmoituksen **Error: Mismatched parameter call!**.

Negatiiviset luvut toimivat ainoastaan vakioille, eli muuttujien tai funktioiden vähennyslaskut/negatiot eivät onnistu.

Viimeisin tulos tallentuu **ans**-nimiseksi funktioksi, mutta sitä ei enää voi käyttää uuden funktion määrittelyssä, esimerkiksi **ans + 3** palauttaa vain **'(ans+3)'**.

Polynomien jakolasku toimii ainoastaan, jos tuloksessa ei ole jakojäännöstä, ts. vain kun jako menee tasan. Joissain tapauksissa polynomin jakolasku voi aiheuttaa **std::logic_error**-poikkeuksen, jota ei käsitellä, johtaen ohjelman kaatumiseen.

5 Tehtävien jako ja aikataulutus

Tehtävät jakautuivat ryhmän kesken suhteellisen tasaisesti, vaikkakin jäsenten aktiiviset kaudet sijoittuivatkin ajallisesti työn eri vaiheisiin. Aikataulu jäi projektin alkuvaiheessa huomattavasti jälkeen, joka kostautui luonnollisesti lopussa. Erityisesti luokkarakenteen asettamat mahdollisuudet kääntyivät haasteiksi ja lopulta rasitteeksi kun aikataulu alkoi painaa päälle. Ryhmän olisi selvästi pitänyt käyttää enemmän aikaa luokkarakenteen ja algoritmien suunnitteluun yhdessä projektin alkuvaiheessa, jotta jokaisen jäsenen itsenäisesti tehdystä työstä olisi saatu paras mahdollinen tulos. Nyt aikaa ja vaivaa kului tarpeettomasti asioiden tekemiseen kahteen kertaan. Toisaalta ryhmän sisäisen viestinnän puutteellisuus johti alkuvaiheessa siihen että jäsenten kesken ilmeni epävarmuutta siitä, minkä osan tulisi olla tehtynä jotta toista voidaan aloittaa. Käytännössä kävi jotakuinkin niin, että parseria ei oltu tehty, sillä luokkarakenne ei ollut valmis eikä parseri näin ollen voinut luoda valmista funktiopuuta. Toisaalta käyttöliittymää ei tehty sillä ilman toimivaa parseria ei ollut paljoa tehtävissä. Luokkarakenne taas oli niin laaja kokonaisuus, ettei sitä voitu saada edes yksinkertaisimpaan muotoonsa muiden osien vaatimassa aikataulussa.

Ryhmän jäsenet käyttivät aikaa seuraavasti:

- Petteri Hyvärinen, 50 tuntia
- Toni Rossi, 40 tuntia
- Ian Tuomi, 60 tuntia

6 Eroavaisuudet alkuperäiseen suunnitelmaan

Laskuoperaatioiden toteutukset tehtiin alkuperäisestä suunnitelmasta poiketen kokonaan **simplify()**-jäsenfunktioihin. Lisäksi funktioiden tyyppin tunnistamista

varten tehtiin yksinkertainen järjestelmä, jossa jokaista funktiotyyppiä vastaa oma kokonaislukunsa.

Alkuperäisestä suunnitelmasta puuttuivat myös täysin muuttujat, joita varten tehtiin oma luokkansa. Tässä luokassa otettiin huomioon myös luvussa 2 mainittu muuttujien nimien törmäysten estäminen.

Alkuperäisessä suunnitelmassa esiintyviä lisäominaisuuksia ei saatu derivoinnin lisäksi tehdyksi.

A Luokkarakenne



Viitteet

- [1] Lamport, Leslie, *A Survey of Gröbner Bases and Their Applications*,
saatavissa <http://www.math.ttu.edu/~ljuan/report.pdf>