SYMBOLIC CALCULATOR – GROUP 1

RYHMÄN JÄSENET:

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

Vaatimusmäärittely

KÄYTTÖLIITTYMÄ

Ohjelman käyttöliittymä on komentorivipohjainen. Jos käyttäjä syöttää pelkän lausekkeen, ohjelma sieventää lausekkeen ja tulostaa tuloksen ruudulle. Tulos tallennetaan ans-muuttujaan, jota voidaan käyttää lausekkeissa funktion tapaan. Käyttäjä voi määritellä omia funktioita komennolla `<funktion nimi> = <lauseke>`, esimerkiksi `f = x*y + 1`. Määritelty funktio tulostetaan ruudulle sievennettynä. Määrittelyn jälkeen funktioita voi käyttää laskuissa.
Funktioiden arvoja yksittäisissä pisteissä voi laskea komennolla `<funktio>[<muuttuja1>=<arvo1>,<muuttuja2>=<arvo2>,..]`, esimerkiksi yllä olevalla f:n määritelmällä lausekkeen `f[x=1,y=2]` arvo olisi 3. Myös funktiolle `f(x)`, `f[7]` toimii odotetulla tavalla ja `f[x^2]` korvaa `x`:n funktiolla `x^2`. Jos yhtäsuuruusmerkkiä ei ole käytetty, on se sama kuin käyttäjä olisi kirjoittanut "`ans = ...`"

SYÖTTEEN PARSIMINEN

Syötteestä tarkistetaan aluksi ettei siinä ole virheellisiä merkkejä. Jos syötteenä on ohjelmalle suunnattu erikoiskomento, esim. "`clear`", "`exit`", "`who`", "`prefix`", "`infix`", "`help`" tms. toimitaan sen mukaisesti. Muussa tapauksessa syöte jaetaan deque-tietorakenteeseen. Sanat ja luvut kuten "omenaPiirakka" ja 123.456 pidetään sellaisenaan. Operaattorit kuten "*" tai "(" erotetaan. Välilyönnit syödään pois.

Seuraavaksi tarkistetaan onko syötteessä yhtäsuuruusmerkkiä, ja jos on, onko sitä käytetty oikein. 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`").

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 Djikstran "Shunting Yard"-algoritmia. Prefix-notaatiota ei käsitellä sen syvällisemmin, siitä vain poistetaan sulut.

Ohjelman rakenne

KÄYTTÖLIITTYMÄ

Käyttöliittymä totetutetaan omaan tiedostoonsa, johon tulee main-funktion lisäksi joitakin apufunktioita. Käyttäjän määrittelemistä funktioista pidetään kirjaa main-funktion sisällä map-rakenteessa, jonka avaimena on funktion nimi merkkijonona ja arvona osoitin funktiopuuhun.

LUOKKARAKENNE

Käsittelemme 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. Laajempiin kokonaisuuksiin päästään ottamalla mukaan transsendenttifunktiot.

Matemaattisen olion tulkitseminen *funktioina* tuo mm. analyysin kautta käyttöömme monia hyödyllisiä työkaluja, joiden avulla symbolisia lauseita voidaan tarkastella ja muokata välittämättä *funktion* määrittely- tai arvojoukoista. Jos esimerkiksi tiedämme *funktion* olevan lineaarinen, voimme jakaa *funktion* tarkastelun useaan pienempään osaan, sillä $f(ax + by) = af(x) + bf(y)$ kaikille lineaarisille *funktioille f* riippumatta siitä mitä x ja y edellisessä lauseessa ovat – ja ennen kaikkea f voi olla muilta ominaisuuksiltaan mielivaltainen, ainoa vaatimus on lineaarisuus. Edellä esitetyn kaltaisista säännöistä muodostuu luonnollisesti hierarkinen luokkarakenne, jota mukailemme myös omassa harjoitustyössämme. Johtuen matemaattisten rakenteiden monimuotoisuudesta valitsemamme luokkarakenne tulee varmasti elämään koko projektin ajan, mutta kantavana ajatuksena on edellä mainittu suurempien kokonaisuuksien pilkkominen hallittavampiin paloihin käyttämällä yleisiä laskusääntöjä.

Function-luokan perii ***NaryFunction***-luokka, josta luokkarakenne haarautuu kolmeen osaan: ***NullaryFunction***, ***UnaryFunction*** sekä ***BinaryFunction***. Tässä lähestymistavassa *funktiot* jaetaan eri haaroihin niiden ottamien *parametrien* lukumäärän mukaan. Tässä vaiheessa projektia olemme jättäneet avoimeksi, tarvitsemmeko esimerkiksi ***TernaryFunction***-tyyppistä luokkaa tai yleisemmin useampia kuin kaksi *parametria* ottavia *funktioita*. On myös otettava huomioon toiset lähestymistavat, kuten edellä sivuttu jako lineaarisiin ja epälineaarisiin funktioihin tai jako aritmeettisiin ja loogisiin operaattoreihin. Voidaan siis todeta, että emme ole kaukana luokkarakenteesta, jossa esiintyy myös moniperintää. *Funktioiden* jaotteluun ariteetin perusteella kannustaa myös tämän mallin käyttökelpoisuus matemaattisen lausekkeen parsimisessa. Ilmoittamalla jokainen alkio lausekkeessa parametrisena *funktiona*, päädytään nopeasti selkeään ja tehokkaaseen puurakenteeseen, jossa solmukohdat muodostuvat unaarisista ja binäärisistä *funktioista*, kuten +, -, * , | ... | (itseisarvo) tai <...,...> (sisätulo) ja lehdet nullaarisista ja unaarisista *funktioista* kuten muuttuja x tai vakio 2. Puurakenteessa esimerkiksi binäärisestä operaatiosta haarautuvia oksia voidaan käsitellä sekä erikseen, toisistaan riippumattomina kokonaisuuksina että toisaalta abstrakteina operaation *parametreina*. Lausekkeesta muodostetulla puulla on aina vain yksi juuri, joten viestien välittäminen kaikille lausekkeessa esiintyville osille onnistuu helposti juuren kautta. Puurakenteen hyötyihin lukeutuu myös rekursiivisesti toteutettavat toiminnot. Esimerkkinä lauseketta evaluoitaessa kutsutaan juurelle jäsenfunktiota ***evaluate(map<string, double> points)***, jossa parametri ***points*** antaa pisteen, jossa *funktion* arvo tahdotaan laskea. Jokainen solmukohta kutsuu alipuilleen samaa jäsenfunktiota, jonka jälkeen palauttaa oman operaationsa arvon, käyttäen

alemmilta tasoilta saatuja arvoja. Laskusääntöjen toteuttaminen on puurakenteessa myös tehokasta, sillä kuten jo aiemmin todettiin, kokonaista alipuuta voidaan käsitellä abstraktina *funktiona*. Otetaan esimerkkinä yhteenlaskun ominaisuus *(a + b) + c = a + (b + c)*. Lausekkeen vasen puoli realisoituisi suunnittelemassamme rakenteessa kaksitasoisena puuna, jossa juuressa on binäärinen operaatio ***Sum***, jolla on parametreinaan vasemmassa haarassa toinen *funktio* ***Sum*** ja oikeassa haarassa funktio *c*. Jälleen vasemman haaran *summafunktion parametreina* ovat *funktiot a* ja *b*. Lausekkeen oikea puoli saadaan kun tehdään uusi puu, jossa ***newSum.lhs = (oldSum.lhs).lhs*** ja ***newSum.rhs = (oldSum.lhs).rhs + rhs***. Tämän pyörittelyn takana piilee tehokkuus, sillä *a, b* ja *c* voivat jälleen olla mitä tahansa *funktioita*, vaikkapa *a = 2 + 5, b = exp(-2 + πN), c = sqrt(42)*. Samalla tavalla voidaan käsitellä erikseen jokainen solmukohta ja siitä haarautuvat oksat, soveltaen aina kyseessä olevan *operaation* ominaisuuksia.

Valitsemamme luokkarakenne toimii jo sinällään mainiosti koko lausekkeeseen kohdistuviin toimintoihin, kuten lausekkeen sieventäminen, evaluointi tai derivointi. Kuitenkaan koko sovellusta ei voida rakentaa pelkkien luokkien varaan, vaan tarvitaan ulkopuolisia funktioita, joihin sijoitetaan edellä esitellyn kaltaiset laskutoimitukset ja niiden laskusäännöistä johtuvat muokkaukset. Pelkästään ***evaluate(...)*** -jäsenfunktion varaan *funktion* toteutusta ei kannata rakentaa, sillä joskus *funktion parametrit* vaikuttavat siihen, mitä lähestymistapaa kannattaisi käyttää. Myös paluuarvo saattaa vaihdella, kuten esimerkiksi tilanteissa *x + a* ja *x + x*, joista ensimmäisessä lauseke ei sievenny joten tuloksena on *summafunktio*, *parametreinaan x* ja *a*. Jälkimmäistä lauseketta sievennettäessä paluuarvo on *kertolaskufunktio*, *parametreinaan* vakio 2 ja muuttuja *x*. Näin ollen jokaisen *funktion* toteutus palauttaa osoittimen ***Function***-tyyppiseen olioon – esimerkkinä geneerinen yhteenlasku ***Function* genericAddition(Function* lhs, Function* rhs)***, jolle tehdään erikoistus ***Function* genericAddition(Function* lhs, Sum* rhs)*** aiemmin esiteltyä tilannetta *a + (b + c)* varten. Operaattoreiden ylikuormittaminen voisi olla yksi vaihtoehto ***genericAddition***-tyylisille globaaleille funktioille, mutta niiden käytössä on ainakin kaksi ongelmaa, nimittäin edellä mainittu paluuarvon riippuvaisuus *parametreista* (pienimmän yllätyksen periaate) sekä se, että ylikuormitettavia operaattoreita on vain rajallinen määrä joten kun kaikki operaattorit on jo käytetty, joudutaan kuitenkin turvautumaan muihin ratkaisuihin. Kumpikaan ongelmista ei ole ylitsepääsemätön, mutta johtaa nopeasti sekavuuksiin, joten olemme päättäneet noudattaa kaikissa tapauksissa samaa rakennetta.

LISÄOMINAISUUDET

Matriisit ja vektorit toteutetaan ***Matrix***-luokkana, joka perii ***NullaryFunction***-luokan. Jos matriisi halutaan sieventää tai evaluoida, kyseistä operaatiota kutsutaan kaikille matriisin alkioille. Matriisien ja vektorien laskuoperaatiot toteutetaan kukin omana luokkanaan, joka perii joko ***UnaryFunction***-luokan tai ***BinaryFunction***-luokan. Todettakoon, että vektoreiden toteutuksen jälkeen voidaan kompleksiluvut toteuttaa helposti kahden mittaisena vektorina, käyttäen hyväksi analogiaa kaksiulotteisen reaalitason ja kompleksitason välillä.

Differentiaaliyhtälöiden suunnittelussa ei ole päästy vielä selkeään luokkarakenteeseen. Aihe on hyvin laaja ja vaatii lisäksi mm. valmiuden ratkaista karakteristisia yhtälöitä (yhtälöiden ratkaisu on yksi lisäominaisuuslistan kohdista). Differentiaaliyhtälöille löytyy useita numeerisia menetelmiä, mutta projektin ollessa symbolinen laskin, tulisi luultavasti valita toisenlainen lähestymistapa, ja kuulisimme mielellämme ehdotuksia tämän ominaisuuden toteutukseen. Luonnollista olisi edetä toteutuksessa yksinkertaisesta monimutkaisempaan, koko ajan laajentaen käytettävissä olevien työkalujen määrää.

Toteutus

PUUNRAKENNUS

Syötteestä rakennetaan abstrakti syntaksipuu kutsumalla yhä uudelleen rekursiivista ***Function* parsePrefix(std::deque<std::string>)***-funktiota. Jokaisen syötteen kohdalla luodaan uusi funktio-objekti jonka lapset lisätään vastaavasti kutsumalla parsePrefix-funktiota päätä lyhyemmällä syötelistalla. Lopussa puu palautetaan. Funktion alkuperäinen kutsuja saa osoittimen puun päähän.

Jos puurakennus on jostain syystä epäonnistunut, virheellisen syötteen takia tai muuten, ilmoitetaan siitä käyttäjälle ja tuhotaan tähän asti rakennettu puu. Kun puu on rakennettu, asetetaan sen kohdefunktion nimi osoittamaan siihen.

Funktioihin kohdistuvat "meta"-komennot kuten simplify täytyy ajaa funktion juuressa. Esim. "`f = simplify g`", "`f = simplify x^1+7`", ovat oikein mutta "`f=5+simplify x^1`" ei ole.

Seuraavassa taulukossa on selvyyden vuoksi esitetty f:llä ja g:llä mitä vain valiidia ennaltamäärättyä funktiota. f:n tilalle voisi myös kirjoittaa minkä vaan valiidin syötteen, (esim. "+ 5 x" tai "5+x", riippuen ohjelman senhetkisestä odotteesta.)

Komento	Selitys	Huomioita
f	Asettaa f:n ans:iin	Sama kuin "ans = f"
f(x) = x^2	Määrittelee f:n funktioksi x:n suhteen.	
g = ans	Asettaa ans:in g:hen	
f(4)	Kutsuu funktiota f arvolla 4.	Asettaa ans:iin arvon 4^2 = 16. ("ans = f(4)")
f(x,y) = x^2+y^2	Kahden muuttujan funktio.	
who	Listaa muuttujat ja funktiot	
simple_f = simplify f	Yrittää sieventää f:n.	
solve(f, 0, x)	Yrittää ratkaista funktion f(x)=0 juuret.	
prefix tai infix	Vaihtaa odotetun syötteen muodon.	Defaulttina infix (esim. (3+6)*x^7
help	Listaa komennot	
clear	Poistaa kaikki tiedot	
exit	Lopettaa ohjelman	

Taulukko 1. Komentoja

LUOKKIEN TOTEUTUKSESTA

Symbolisessa laskennassa ei lasketa lausekkeiden numeerisia arvoja, vaan keskitytään lausekkeen rakenteeseen ja sen osasten vaikutuksiin toisiinsa nähden. Tämä näkyy myös luokkien toteutuksissa, joissa itse operaatio jota *funktio* kuvaa, voi olla triviaali, mutta sen suhde muihin operaatioihin ratkaiseva. Siksi *funktio*-oliot sisältävät enemmänkin tietoa *funktion* ominaisuuksista, toteutusten sijaitessa itseasiassa luokan ulkopuolella. Myös puurakenne hoidetaan *funktioiden* kautta.

Seuraavassa taulukossa on esitelty kaksi luokkaa luokkarakenteen juuresta. Nämä ovat abstrakteja

luokkia, joista käy hyvin ilmi luokkarakenteemme logiikka. Liitteessä A on esitelty lyhyt koodinpätkä joka havainnollistaa luokkien käsittelyä summalausekkeessa.

LUOKKA	***JÄSEN***	***KUVAUS***
Function	`std::set<std::string> _variables (private)`	*_variables* sisältää 0 ... n kpl string-arvoja. Nämä arvot ovat *funktion* muuttujien nimiä, esimerkiksi lausekkeesta *f(x) = x*y* muodostetulla *funktiolla* on *_variables*-jäsenessään yksi arvo, "*x"*. Vakiofunktioilla *_variables* on tyhjä.
Function	`virtual Function* expand() const (public)`	Jäsenfunktio, jonka tarkoituksena on "levittää" *funktion* esittämä lauseke, esimerkiksi laskemalla sulkulausekkeet auki. *Expand()* ei muokkaa puurakennetta, vaan luo uuden puun, joka sisältää levitetyn lausekkeen esityksen.
Function	`Virtual Function* simplify() const (public)`	Yksinkertaistaa lausekkeen. Samoin kuin *expand()*, luo uuden puun eikä muokkaa puurakennetta.
Function	`Virtual Function* evaluate( std::map<std::string, long> variable_values) const (public)`	Laskee *funktion* arvon *variable_values*:in ilmoittamassa pisteessä. *variable_values*:issa key on muuttujan nimi ja value on muuttujan arvo. *variable_values* voi sisältää mitä tahansa muuttujan nimiä, mutta *evaluate(...)* käyttää vain niitä, jotka sillä on *_variables*-setissä. *variable_values*:in ei välttämättä tarvitse toisaalta sisältää kaikkia *_variables*-setin muuttujia, vaan esimerkiksi lausekkeesta *f(x, y) = 2*x + 5*y* tulee *4 + 5*y* kun *variable_values* sisältää ainoastaan parin ("x", 2).
Function	`virtual const std::set<std::string> getVariables() const (public)`	Palauttaa *_variables* jäsenen.
Function	`virtual Function* clone() const (public)`	Muodostaa uuden olion, joka on kopio tästä. Tarvitaan mm. geneerisissä laskutoimitusten toteutuksissa, joissa operaatioita pilkotaan ja muodostetaan uusia puita.
NaryFunction : public Function	`std::vector<Function*> _operands (private)`	Vektori tämän *funktion* operandeista. Operandeja voi olla 0 ... n kpl.
NaryFunction : public Function	`Virtual void addOperand(Function* child) (public)`	Lisää operandin *funktioon.*
NaryFunction : public Function	`Virtual const std::vector<Function*> getOperands() const`	Palauttaa operandit.

	(public)	

Taulukko 2. Luokkarakenteen esittelyä

Työnjako

Jokainen ryhmän jäsen vastaa omista aihealueistaan, mutta luonnollisesti kaikki voivat osallistua suunnitteluun ja toteutukseen. Lisäominaisuuksien toteutuksessa henkilö itse määrittelee toivotun syötteen muodon sekä käyttöliittymälle näkyvän rajapinnan omiin luokkiin.

Ian Tuomi

- Syötteen parsiminen
- Mielivaltaisen tarkkuuden luvut ja niiden operaatio

Toni Rossi

- Käyttöliittymä
- Vektori- ja matriisilaskenta

Petteri Hyvärinen

- Luokkarakenne
- Differentiaaliyhtälöt

Testaus

Käyttöliittymän ja parserin toimivuus testataan erilaisilla valideilla ja virheellisillä syötteillä. Itse laskutoimitusten oikeellisuuden todentaminen on verrattain helppoa, ja sen tulisi tapahtua iteratiivisesti kehitystyön aikana.

Aikataulu

14.11. Käyttöliittymä ja syötteen parsiminen toimivat minimivaatimusten mukaisille syötteille, eli polynomeille ja niiden laskutoimituksille.

28.11. Lisäominaisuudet toimivat

29. - 3.12. Testaaminen/debuggaaminen

Liite A

GenericAddition toteutus:

```
// In the most general case we just return a Sum object.
// If there is a more specific overload, that will be used instead.
Function* genericAddition(const Function* lhs, const Function* rhs)
{
    return new Sum(lhs->clone(), rhs->clone());
}

// When adding together a function and a sum, we know (since + is commutative),
// that we can interchange the places of function and the first or second term of
// the sum, i.e. (a + b) + c = a + (b + c) = (b + c) + a = b + (c + a)
Function* genericAddition(const Function* function, const Sum* sum)
{
    return genericAddition(sum->getLhs(), genericAddition(sum->getRhs(), function));
}

Function* genericAddition(const Sum* sum, const Function* function)
{
    return genericAddition(function, sum);
}

Function* genericAddition(const Sum* lhs, const Sum* rhs)
{
       return genericAddition(lhs->getLhs(),
                           genericAddition(lhs->getRhs(), rhs));
}

Function* genericAddition(const ConstantFunction* lhs, const ConstantFunction* rhs)
{
    if (lhs->isNumber() && rhs->isNumber())
        return new ConstantFunction(lhs->getNumber() + rhs->getNumber());
    else
    {
        if (lhs->getSymbol() == rhs->getSymbol())
            return new Multiplication(
             new ConstantFunction(lhs->getNumber() + rhs->getNumber()),
             lhs->getSymbol());
        else
            return new Sum(lhs->clone(), rhs->clone());
    }
}
```