TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Antti Veisu

Lukiomatematiikan tehtävien automaattinen tarkastus testiarvoja kokeilemalla

Diplomityö

Tarkastaja: professori Antti Valmari
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 5. syyskuuta 2012

TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Tietotekniikan koulutusohjelma
Veisu, Antti: Lukiomatematiikan tehtävien automaattinen tarkastus testiarvoja kokeilemalla
Diplomityö, 46 sivua
Lokakuu 2012
Pääaine: Ohjelmistotiede
Tarkastaja: professori Antti Valmari
Avainsanat: automaattinen tarkastaminen, CAA, matematiikan automaattinen tarkastaminen, lukiomatematiikan tehtävät

Tietotekniikan rooli opetuksessa kasvaa ja erityisesti matematiikan opetukseen se tarjoaa mielenkiintoisia sovelluksia. Yksi sellainen on matemaattisten tehtävien automaattinen tarkastus. Matematiikan tarkka ja looginen luonne mahdollistaa matematiikan tehtävien koneellisen tarkastamisen syvällisemmän kuin monissa muissa oppiaineissa. Matemaattisten tehtävien automaattiseen tarkastukseen on olemassa useita ohjelmia, mutta monet niistä on tarkoitettu lähinnä opettajan avuksi arviointiin. Tässä diplomityössä tarkastellaan opiskelijan avuksi tarkoitetun web-pohjaisen tarkastusohjelman suunnittelua ja toteutusta. Ohjelma rajoittuu lukiomatematiikan ensimmäisille kursseille tyypillisiin lasku- ja yhtälönratkaisutehtäviin. Laskutehtävissä matemaattinen lauseke saatetaan haluttuun muotoon ja yhtälönratkaisutehtävissä yhtälö tai epäyhtälö ratkaistaan jonkin muuttujan suhteen.

Ohjelman erityispiirteenä on, että sille voi syöttää välivaiheita. Tehtävien vastaukset ovat päättelyketjuja, joiden askeleet, eli välivaiheet, ohjelma tarkastaa. Virheellisen välivaiheen löytäessään ohjelma osoittaa sen käyttäjälle. Se on ohjelman pääasiallinen tapa auttaa tehtävien ratkaisussa. Vastaukset syötetään tekstimuodossa noudattaen merkintätapaa, joka suunniteltiin muistuttamaan mahdollisimman paljon paperilla laskemista. Ohjelman toinen erityispiirre on, että se tarkastaa vastaukset kokeilemalla testiarvoja vastauksen muuttujille. Menetelmän hyöty on, että virheen löydyttyä käyttäjälle voidaan antaa esimerkkiarvot, joilla vastaus on väärin.

Välivaiheet sisältävän merkintätavan laatiminen ja sitä tulkitsevan ohjelman osan toteuttaminen osoittautui melko suoraviivaiseksi tehtäväksi. Testiarvojen kokeilemiseen perustuva tarkastus ei ole täysin varma, mutta testaus osoitti, että ohjelma löytää käytännössä suurimman osan virheistä. Osoittautui, että laskutehtävissä testiarvot voidaan valita hyvin vapaasti. Pienikin virhe tyypillisesti muuttaa lausekkeen arvoa merkittävästi kaikilla muuttujien arvoilla, jolloin ohjelma löytää virheen helposti. Yhtälönratkaisutehtävissä puolestaan testiarvot pitää valita huolella ja tehtävän oikea ratkaisu pitää tuntea, jotta voidaan valita testiarvot, joilla virheitä löydetään.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master's Degree Programme in Information Technology
Veisu, Antti: Automated Assessment in High School Mathematics by Testing Variable Values
Master of Science Thesis, 46 pages
October 2012
Major: Computer Science
Examiner: Professor Antti Valmari
Keywords: automatic checking, CAA, Computer Aided Assessment, automatic checking in mathematics, high school mathematics exercises

The role of information technology continually grows in teaching, and especially in mathematics it opens up new possibilities. One such possibility is automatic checking of mathematics exercises. The exact and logical nature of mathematics makes it possible to automatically check mathematics exercises more thoroughly than exercises in other disciplines. There are many programs for automatic assessment of mathematics exercises, but many of them are meant mostly to help teachers in assessing the skills of students. This thesis studies the design and implementation of a web-based exercise checking program for aiding students in mathematics. The program is limited to straightforward calculation and equation exercises common in high school mathematics. In calculation exercises a mathematical expression is reduced to a certain form. In equation exercises an equation or an inequality is solved with respect to a certain variable.

An important feature of the program is that it allows the use of intermediate results. The answers to the exercises are chains of deductions, where the steps are the intermediate results, that the program checks. When an erroneous intermediate result is found, the program points it out, which is the main way of helping the user to solve the exercise. The answers are entered as text with a notation that follows traditional pen and paper notations as closely as possible. A second essential feature is that the checking is done by testing different variable values. The advantage of this method is that test values that demonstrate an error can be shown to the user.

Designing the notation including intermediate results and implementing the part of the program that interprets it proved to be quite straightforward. Checking by testing variable values is not a fully reliable method, but experiments with the program showed that in practice it finds most errors. Even a small error typically changes the values of an expression significantly, which makes checking calculation exercises easy. In equation exercises the test values have be chosen carefully. Also, the correct solution to the equation must be available to the program for effective checking of equation exercises.

ALKUSANAT

Haluan esittää kiitokset diplomityön tarkastajalle professori Antti Valmarille tämän aiheen ehdottamisesta sekä neuvoista ohjelman toteuttamisessa ja diplomityön kirjoittamisessa. Kiitokset myös työn ohjaajalle FT Antti Saariselle lausunnon antamisesta.

12.9.2012

Antti Veisu

SISÄLLYS

1. JOHDANTO

Tietotekniikan kehittyessä sen hyväksikäyttö opetuksessa on noussut huomattavan mielenkiinnon kohteeksi [1]. Erityisesti matematiikan opetukseen tietotekniikka saattaa tarjota merkittäviä apuvälineitä, koska tietotekniikka soveltuu hyvin matematiikan tarkkaan ja loogiseen sovellusympäristöön. Eräs tällainen sovellus on matemaattisten tehtävien automaattinen tarkastaminen. Automaattisen tarkastuksen avulla saadaan opettajien työtaakkaa pienennettyä ja tarjottua opiskelijoille työkaluja avustamaan opiskelussa. Erityisesti lukiomatematiikan ensimmäisillä kursseilla suuri osa harjoitustehtävistä on tyypillisesti suoraviivaisia lasku- ja yhtälönratkaisutehtäviä, joiden tarkastus on suhteellisen helposti automatisoitavissa.

Ohjelmia, jotka auttavat tehtävien tarkastuksessa ja arvostelussa, kutsutaan CAA-ohjelmiksi (eng. Computer Aided Assessment) oppiaineesta riippumatta. Matemaattisia CAA-ohjelmia on olemassa useita, kuten esimerkkiksi STACK [2], AIM [3] ja kaupallinen Maple TA [4]. Näiden ohjelmien perusperiaate on, että tehtävä esitetään käyttäjälle ja käyttäjä syöttää tehtävän vastauksen, jonka ohjelma sitten tarkastaa. Vastaus on tyypillisesti matemaattinen lauseke, jonka numeerinen tai algebrallinen yhtäpitävyys tarkastetaan ohjelman tiedossa olevan oikean vastauksen kanssa. Monet näistä ohjelmista tukevat muunlaisiakin tehtäviä, kuten esimerkiksi monivalintatehtäviä.

Tässä diplomityössä tarkastellaan kuinka lukiomatematiikan lasku- ja yhtälönratkaisutehtäville voidaan suunnitella ja toteuttaa ohjelma, joka avustaa niiden tekemisessä. Ohjelman erityispiirre on, että sille annettavissa vastauksissa välivaiheet voivat olla mukana ja ne myös tarkastetaan. Virheellisen välivaiheen löytäessään ohjelma osoittaa käyttäjälle kohdan, jossa virhe sattui. Välivaiheiden paikkoja vastauksessa ei määrätä ennalta eikä niiden lukumäärää rajoiteta. Kirjoittajan tiedossa ei ole toista matematiikan CAA-ohjelmaa, joka tekisi saman.

Ohjelman toinen erityispiirre on, että se tarkastaa tehtävät kokeilemalla testiarvoja tehtävässä esiintyville muuttujille. Virheen löytyessä ohjelma voi siis antaa käyttäjälle suoraan esimerkkiarvot, joilla vastaus ei päde. Tämä tarkastusperiaate ei tosin ole täysin varma, joten ohjelma ei sovellu arvostelun apuvälineeksi. Suurin osa muista matematiikan CAA-ohjelmista perustuu symboliseen laskentaan, joka puolestaan pystyy takaamaan varman tarkastuksen. Ohjelman onkin tarkoitus toimia nimenomaan opiskelijoiden apuvälineenä eikä opettajien apuna arvostelussa toisin

kuin monet muut matematiikan CAA-ohjelmat.

Luvussa 2 käydään läpi ohjelman yleiset vaatimukset ja tavoitteet yksityiskohtaisemmin. Erityisesti ohjelman tukemat lasku- ja yhtälönratkaisutehtävät määritellään. Luvussa 3 esitetään tehtävien vastauksien tarkka syntaksi ja kuinka se tarkastetaan. Luvussa 4 tarkastellaan ohjelman toteutusta korkealla tasolla ja luvuissa 5 ja 6 puolestaan tarkastellaan tehtävien tarkastuksen toteutusta ja käyttöliittymää tarkemmin. Luku 7 selvittää ohjelman nopeuskriittisimmät osat ja tarkastelee jatkokehitysideoita tehokkuuden parantamiseksi. Lopuksi luvussa 8 kerätään yhteen tämän diplomityön tärkeimmät havainnot ja tulokset sekä verrataan ohjelman heikkoja ja hyviä puolia jo olemassa olevien CAA-ohjelmien kanssa.

2. HARJOITUSOHJELMAN VAATIMUKSET

2.1 Yleiset vaatimukset ja tavoitteet

Tämän diplomityön tarkoitus on tutkia kuinka voidaan toteuttaa ohjelma, joka avustaa suoraviivaisten lasku- ja yhtälönratkaisutehtävien harjoittelussa. Tarkemmin ottaen ohjelman tulee tarkastaa käyttäjän syöttämä laskutoimitus tai yhtälönratkaisun päättelyketju ja näyttää virheen paikka, jos löytää sellaisen. Tarkastettavat vastaukset voivat liittyä ohjelman itse antamaan tehtävään, jolloin ohjelman käytössä on myös tehtävän laatijan syöttämää informaatiota, joka avustaa ohjelmaa tarkastuksessa. Tarkastuksen perusperiaatteena on asettaa vastauksen muuttujille testiarvoja ja kokeilla päteekö vastaus. Ohjelmalta ei vaadita täydellistä varmuutta tarkastuksessa vaan pyrkimys on vain avustaa tehtävien ratkaisussa.

Yksi tärkeimmistä ohjelman vaatimuksista on, että käyttäjä voi syöttää vastauksensa muodossa, joka on mahdollisimman lähellä perinteistä paperilla laskemista. Käytännössä tämä tarkoittaa, että vastaus annetaan tekstinä noudattaen ohjelman määräämää syntaksia. Tämä ei välttämättä ole paras mahdollinen käyttöliittymä käytettävyyden kannalta mutta on yksinkertaisesti toteutettavien käyttöliittymien joukossa parhaita. Esimerkiksi harjoitusohjelma, joka antaa rajatun määrän vaihtoehtoja tehtävän ratkaisuun, saattaa houkutella käyttäjää yksinkertaisesti kokeilemaan kaikkia vaihtoehtoja kunnes löytää oikean. Tämä ei tietenkään auta käyttäjää tehtävän ymmärtämisessä. Tekstipohjainen käyttöliittymä selvästikin välttää tämän ongelman.

Tekstipohjaisessa käyttöliittymässä haasteena on tietenkin laatia ja toteuttaa sopiva syntaksi. Sen täytyy olla tarpeeksi ilmaisuvoimainen ja muistuttaa mahdollisimman paljon yleisiä matemaattisia merkintätapoja, että sen voi omaksua nopeasti. Vaarana on, että käyttäjällä menee enemmän aikaa vastauksen syntaksin korjaamiseen kuin itse laskun käsittelyyn. Samalla sen täytyy olla tarpeeksi yksinkertainen, jotta ohjelma on kohtuullisella vaivalla toteutettavissa.

Ohjelma tulee toteuttaa web-sovelluksena, jotta se olisi erittäin helppo ottaa käyttöön. Sen käyttämiseksi tarvitaan vain selain, eikä käyttäjän tarvitse asentaa mitään omalle koneelleen. Lisäksi tekstipohjainen käyttöliittymä on niin yksinkertainen, että se on helppo toteuttaa web-sivuna.

Ohjelman laajuuden rajaamisessa käytetään apuna lukion pitkän matematiikan polynomilaskennan kurssikirjaa [5]. Ohjelman tulisi siis selvitä tämän kirjan ta-

soisten lasku- ja yhtälönratkaisutehtävien tarkastamisesta. Ohjelman muiden vaatimusten kannalta on riittävää, että sen avulla voitaisiin suorittaa noin kymmenen samanaikaisen käyttäjän koekäyttö. Esimerkiksi tehokkuudesta tingitään ajoittain, jos tehokkaamman ratkaisun toteuttaminen on selvästi hankalampaa. Tämän ohjelman on tarkoitus toimia prototyyppinä, jolla voitaisiin testata onko tällaisesta ohjelmasta oikeasti hyötyä.

Kuten edellä mainittiin, ohjelman täytyy tarkastaa kahdenlaisia tehtäviä: laskuja yhtälönratkaisutehtäviä. Tehtävätyypit kuvataan tarkemmin omissa aliluvuissaan. Nämä kaksi tehtävätyyppiä on valittu sillä perusteella, että ne ovat kaikkein yleisimpiä tehtäviä kirjassa [5]. On myös melko itsestään selvää, että tällaisten tehtävien ratkaiseminen on matematiikan perustaitoja. Muuntyyppisiä tehtäviä, joiden tukeminen olisi hyödyllistä, on paljon, mutta tämän diplomityön puitteissa ei voida kovin monenlaisia tehtäviä tukea. Uusien tehtävätyyppien helppo lisääminen tulisi kuitenkin ottaa huomioon ohjelman rakenteessa.

Kumpaakin tehtävätyyppiä on kahta eri muotoa. Määritellyissä tehtävissä käyttäjän syöte liittyy tehtävään, jonka ohjelma on antanut käyttäjälle, jolloin ohjelmalla on käytössään sen oikea vastaus ja mahdollisesti muutakin informaatiota. Vapaissa tehtävissä ohjelma pyrkii tarkastamaan käyttäjän laskutoimituksen tai yhtälönratkaisun ilman mitään lisäinformaatiota. Määritellyt tehtävät ovat tietenkin helpompia ohjelman kannalta, koska ylimääräinen informaatio mahdollistaa varmemman tarkastuksen, mutta vapaat tehtävät tuovat käyttäjälle paljon uusia mahdollisuuksia. Vapaat tehtävät mahdollistavat ohjelman käytön paljon monipuolisemmissa tilanteissa ja säästävät tehtävien laatijan työtä. Määriteltyjen tehtävien lisäystä, muokkausta tai poistoa ei toteuteta web-käyttöliittymään, vaan tehtävien laatija tekee sen muokkaamalla ohjelman web-palvelimella sijaitsevia tehtävien määrittelytiedostoja.

Kokonaisuudessaan ohjelman tulee tukea neljää eri tehtävätyyppiä. Kaikissa tehtävätyypeissä voi käyttää kokonaislukuja ja yhden kirjaimen symboleita muuttujina. Rajoitus kokonaislukuihin on tehty, koska kirjan [5] tehtävissä ei tarvittu desimaalilukuja ja murtoluvut voidaan esittää käyttämällä jakolaskua. Desimaalilukujen käyttö tuskin tuo pedagogisesti mitään uutta arvoa, mutta niiden helppo lisääminen ohjelmaan olisi kuitenkin hyvä ottaa huomioon. Kaikissa tehtävätyypeissä tulee tukea ainakin yhteen-, vähennys- ja kertolaskuja sekä potenssiin korotusta, itseisarvon merkintää ja tietenkin sulkuja. Potenssiin korotuksessa sallitaan myös murtopotenssit, jolloin juuria käsittelevienkin vastausten tarkastus onnistuu. Jakolasku todettiin suunnitteluvaiheessa niin hankalaksi, että se päätettiin toteuttaa vain, jos siihen riittää aikaa. Lopulta sekin saatiin toteutettua aikataulun puitteissa. Nämä laskuoperaattorit riittävät kirjan [5] tehtävien kannalta, mutta uusien laskuoperaattoreiden lisäämisen tulisi kuitenkin olla helppoa ohjelmistotekniseltä kannalta.

Vastausten syntaksin perusperiaatteena on päättelyketjuista muodostuva rakenne. Monien suoraviivaisten lasku- ja yhtälönratkaisutehtävien vastaus on esitettävissä päättelyketjuna ja sellaisina ne perinteisesti esitetäänkin paperilla laskettaessa. Ohjelman täytyy tosin vaatia käyttäjää noudattamaan tarkkaa syntaksia, joka tuo rakenteen eksplisiittisesti esille. Tämä voi muodostua ongelmaksi, koska jotkin käyttäjät eivät välttämättä edes miellä matemaattisten tehtävien ratkaisemista päättelyketjuksi.

Vastaavien rakenteiden käyttöä opetuksessa on jo itse asiassa tutkittu. On olemassa *rakenteiset päättelyketjut* [6, 7] nimellä kulkeva matemaattisten todistusten laatimismenetelmä ja merkintätapa, jota on kokeiltu käytännön opetuksessa. Tulokset ovat olleet positiivisia ja antavat jopa viitteitä, että rakenteisten päättelyketjujen käyttö parantaa oppimista tietyissä tilanteissa [8, 9, 10]. Rakenteiset päättelyketjut ovat uusi matemaattinen notaatio, jolla pyritään tuomaan eksplisiittisesti esille tehtävän ratkaisun tai matemaattisen todistuksen logiikkaa. Vaikka tämän ohjelman syntaksin on tarkoitus muistuttaa perinteisiä paperilla laskemisen merkintätapoja uuden notaation sijasta, molemmat perustuvat pohjimmiltaan päättelyketjusta muodostuvaan rakenteeseen. Rakenteisten päättelyketjujen tutkimustulosten valossa päättelyketjuihin pohjautuvan syntaksin valinta on perusteltua.

2.2 Laskutehtävät

Laskutehtävällä viitataan tehtävään, jonka vastaus on suoraviivainen lasku, jossa numeerista lauseketta muokataan toiseen muotoon askel askeleelta. Alla esitetään muutama esimerkki tällaisista laskutoimituksista.

$$|6-8|^2 = 2^2 = 4$$
$$(x-4)^2 + x - 4 = x^2 - 8x + 16 + x - 4 = x^2 - 7x + 12$$
$$(a+b)^2 = a^2 + 2ab + b^2$$

Ohjelman tulee tarkastaa jokainen yhtäsuuruus parhaansa mukaan ja virheen löytäessään osoittaa käyttäjälle virheellinen yhtäsuuruus. Käytettävyyden kannalta on myös tärkeää, että virheistä laskun alkupäässä ilmoitetaan ensin. Jos vastauksessa on kaksi virhettä ja ensimmäisenä ilmoitetaan jälkimmäisestä, voi käyttäjä tuhlata aikaansa korjatessa virhettä, jolla ei lopulta ole merkitystä.

Määritellyissä laskutehtävissä tehtävän laatija määrittelee ohjelmalle lopullisen lausekkeen, johon käyttäjän pitää päätyä. Tehtävänanto voidaan määritellä yksinkertaisesti aloituslausekkeella, josta lähdetään liikkeelle. Käyttäjälle annetaan tämä aloituslauseke ja tehtävän laatijan syöttämä tekstuaalinen ohje mihin muotoon lauseke pitää muokata. Vaihtoehtoisesti aloituslausekkeen voi jättää määrittelemättä, jolloin tarkoitus on antaa kaikki tarvittava informaatio käyttäjälle sanallisessa

ohjeessa. Tämän ominaisuuden tarkoitus on antaa ainakin yksinkertainen tuki sellaisiin tehtäviin, joissa aloituslausekkeen muodostus on osa tehtävän ratkaisua. Lisäksi ohjelmalle on määriteltävä jokaisessa tehtävässä käytetyt muuttujat ja niiden arvoalueet. Arvoalueet voivat muodostua lukusuoran pisteistä ja väleistä, jotka voivat olla avoimia, puoliavoimia tai suljettuja. Arvoalueet voivat olla myös äärettömiä kummastakin päästä.

Vapaissa laskutehtävissä ei tarkastuksessa ole mitään ylimääräistä informaatiota käytettävissä, joten ohjelma voi vain tarkastaa käyttäjän syöttämien yhtäsuuruuksien oikeellisuuden. Niissä voi käyttää mitä tahansa muuttujia ja niiden arvoalueeksi oletetaan koko lukusuora.

2.3 Yhtälönratkaisutehtävät

Yhtälönratkaisutehtävällä viitataan tehtävään, jossa looginen lauseke pitää ratkaista jonkin muuttujan suhteen. Tehtävän vastaus on päättelyketju, jossa lauseketta muokataan kohti yhtäpitävää ratkaistua muotoa. Looginen lauseke voi sisältää vertailuoperaattoreita $=, \neq, <, \leq, \geq$, ja $>$ ja loogisia operaattoreita $\wedge$ ja $\vee$. Muille loogisille operaattoreille ei ilmennyt tarvetta kirjan [5] tehtävissä. Vaikka tällaiset tehtävät voivat itse asiassa olla myös epäyhtälönratkaisua, viitataan tähän tehtävätyyppiin selkeyden vuoksi vain yhtälönratkaisutehtävinä. Alla on muutamia esimerkkejä tällaisista päättelyketjuista yleistä merkintätapaa käyttäen.

$$2(x-1) = 4x \iff 2x - 2 = 4x \iff -2x = 2 \iff x = -1$$
$$3a^2 - 12 > 0 \iff 3a^2 > 12 \iff a^2 > 4 \iff |a| > 2 \iff a < -2 \vee a > -2$$
$$2(x-y) = 3 \iff 2x - 2y = 3 \iff x = y + 3/2$$

Tämä rakenne vastaa täysin laskutehtävien rakennetta, kun loogiset lausekkeet korvaa numeerisilla lausekkeilla ja yhtäpitävyydet yhtäsuuruudella. Yhtälönratkaisussa kynällä ja paperilla voi ja joutuu tosin tekemään paljon monimutkaisempiakin päättelyitä. Yksi tärkeimpiä rakenteita on mahdollisuus haaroittaa päättelyketju joidenkin ehtojen mukaan. Kirjan [5] tehtävissä tätä rakennetta ei itse asiassa tarvita, mutta rakenne on muuten niin tärkeä, että sen tukemista vaaditaan ohjelmalta. Alla on yksinkertainen esimerkki yhtälönratkaisusta, jonka täsmällistä ratkaisua ei voi edes esittää ilman tätä rakennetta.

$$ax \leq 0 \iff \begin{cases} x \geq 0, \text{kun } a \leq 0 \\ x \leq 0, \text{kun } a \geq 0 \end{cases}$$

Haarautumisrakenteen täytyy toimia myös rekursiivisesti eli päättelyketjun haara voi haarautua uudestaan. Tämän rakenteen ilmaisuvoimaa joudutaan kuitenkin

rajoittamaan, jotta ohjelman toteutus ei mene liian hankalaksi, mihin palataan aliluvussa 3.4. Ehtojen täytyy olla sellaista muotoa, että ne määrittelevät yhdelle muuttujalle arvoalueen, joka on suoraan jäsennettävissä ehdon esitysmuodosta. Lisäksi yhden haarauman kaikkien ehtojen pitää koskea vain yhtä muuttujaa. Tämän takana on ajatus siitä, että jokainen haara kattaa yhtälön ratkaisun, kun haarauman muuttuja on annetulla arvoalueella.

Samaan tapaan kuin laskutehtävissä myös yhtälönratkaisutehtävissä ohjelman tulee tarkastaa jokainen yhtäpitävyys osoittaen virheelliset yhtäpitävyydet. Päättelyketjun haaraumassa täytyy yhtäpitävyys tietenkin tarkastaa jokaisen haaran ensimmäisen lausekkeen kanssa erikseen. Ohjelman tulee myös tarkastaa, että haarat kattavat yhdessä muuttujan koko arvoalueen. Virheiden ilmoitusjärjestyksessä pätee samat vaatimukset kuin laskutehtävissä.

Määritellyissä yhtälönratkaisutehtävissä tehtävän laatija määrittelee tehtävän ratkaisun ohjelmalle, jotta tehtävä voidaan tarkastaa paremmin. Aivan vastaavasti kuin laskutehtävissä tehtävänannon voi määritellä täysin sanallisessa muodossa tai aloituslausekkeena sanallisen ohjeen kanssa. Näiden lisäksi ohjelmalle pitää määritellä tehtävässä käytettävät muuttujat, niiden arvoalueet ja muuttuja, jonka suhteen tehtävä pitää ratkaista. Muuttujien arvoalueet ovat samanlaisia kuin laskutehtävissä.

Vapaissa yhtälönratkaisutehtävissä ei edes tiedetä minkä muuttujan suhteen yhtälöä yritetään ratkaista, joten käyttäjän syötteen tarkastaminen on paljon rajallisempaa. Ohjelman tulee vain tarkastaa parhaansa mukaan siinä olevat yhtäpitävyydet. Samaan tapaan kuin vapaissa laskutehtävissä, näissä tehtävissä muuttujien arvoalueeksi oletetaan koko lukusuora.

3. TEHTÄVIEN TARKASTUS TESTIARVOJA KOKEILEMALLA

3.1 Testiarvoilla kokeilemisen periaate

Kuten aliluvuissa 2.2 ja 2.3 kävi ilmi, vastausten tarkastuksen ytimen muodostaa kahden lausekkeen välisen relaation pätevyyden tarkastus. Laskutehtävissä relaatio on yhtäsuuruus kahden numeerisen lausekkeen välillä ja yhtälönratkaisutehtävissä yhtäpitävyys kahden loogisen lausekkeen välillä.

Kahden lausekkeen välisen relaation tarkastus suoritetaan yksinkertaisesti antamalla lausekkeen muuttujille arvoja, evaluoimalla lausekkeet ja katsomalla päteekö relaatio. Tarkemmin ottaen jokaiselle muuttujalle määrätään n testiarvoa. Näistä testiarvoista sitten muodostetaan kaikki mahdolliset kombinaatiot, joilla lausekkeet evaluoidaan. Menetelmä on periaatteeltaan hyvin yksinkertainen ja on helposti laajennettavissa monenlaisille relaatioilla. Kun tarkastus toteutetaan hyvin, uuden relaation tarkastuksen voi lisätä toteuttamalla yksi uusi funktio, joka palauttaa päteekö uusi relaatio kahden arvon välillä. Toisena hyvänä puolena se tuottaa käyttäjälle selkeää palautetta, kun testiarvot, joilla relaatio ei päde, ilmoitetaan käyttäjälle virheen löytyessä.

Yksi menetelmän huono puoli on, että se ei takaa varmaa tulosta, mutta se onkin hyväksyttävää tämän ohjelman kohdalla. Tässä luvussa nähdään kuitenkin vielä, että tietyn tyyppisissä tehtävissä menetelmä on varma ja muissakin tehtävissä suurin osa virheistä löydetään kunhan muuttujien testiarvot valitaan huolella.

Toinen menetelmän huono ominaisuus on sen ajankulutus suurilla määrillä muuttujia. Kun muuttujia on s kappaletta ja jokaiselle on n testiarvoa, on testiarvojen kombinaatioita n^s. Tarkastuksen ajankulutus on siis kertaluokassa $O(n^s)$, kun lausekkeiden evaluointiin ja evaluoitujen arvojen tarkastukseen oletetaan kuluvan vakiomäärä aikaa. Tästä nähdään selvästi, että testiarvojen ja erityisesti muuttujien lukumäärä vaikuttaa merkittävästi tarkastuksen ajankulutukseen. Suurimmassa osassa kirjan [5] tehtäviä on kuitenkin vain yksi tai kaksi muuttujaa, joten se ei ole suuri ongelma. Lausekkeiden evaluointiin kuluva aika riippuu pääasiassa lausekkeiden pituudesta. Lausekkeet ovat ihmisen kirjoittamia, joten käytännössä lausekkeet eivät ole kovin pitkiä. Tämän takia tarkastuksen ajankulutusta kertaluokkatasolla tarkasteltaessa ei ole järkevää yrittää ottaa huomioon lausekkeiden evaluointiin

kuluvaa aikaa. Evaluoitujen arvojen tarkastuksen hoitavat funktiot puolestaan kuluttavat vakiomäärän aikaa yhtäsuuruuden ja yhtäpitävyyden tapauksissa.

Useimmat matematiikan CAA-ohjelmat toimivat symbolisen laskennan menetelmillä [11]. Niiden käytössä on monta etua verrattuna testiarvoilla kokeilemiseen. Ennen kaikkea kahden lausekkeen yhtäsuuruus voidaan tarkastaa varmasti, koska symbolinen laskenta perustuu lausekkeiden muokkaamiseen tunnettuja matemaattisia identiteettejä hyväksi käyttäen. Lisäksi symbolinen laskenta sopii monipuolisempiin tehtäviin. Esimerkiksi tehtävien, joissa käyttäjän tehtävä on syöttää omavalintainen jonkin tietyn ominaisuuden omaava matemaattinen objekti, toteutus onnistuu CAS-järjestelmillä [12]. Esimerkkinä tällaisesta objektista voisi olla vaikka parillinen funktio. Kääntöpuolena on, että symbolinen laskenta on hyvin monimutkaista. Monet ohjelmat turvautuvatkin valmiiksi toteutettuihin symbolisen laskennan järjestelmiin, eli CAS-järjestelmiin (eng. Computer algebra system).

3.2 BNF-notaatio ja numeeristen lausekkeiden syntaksi

Tässä luvussa esitetään myös syntaksi, jota tarvitaan ohjelman käyttämiseksi. Tässä luvussa se pyritään esittämään selkeässä ja helposti opittavassa muodossa mutta varsinainen toteutus pohjautuu eri muodossa olevaan kielioppiin numeeristen ja loogisten lausekkeiden kohdalla. Osittain samaa syntaksia käytetään sekä tehtävien vastauksissa että tehtävien määrittelytiedostoissa. Perusperiaatteena koko ohjelman syntaksissa on, että välilyönneillä, rivivaihdoilla ja muilla tyhjillä merkeillä ei ole merkitystä, kunhan tekstialkiot erottuvat toisistaan. Tämä linjaus tehtiin lähinnä siksi, että vastaukset ja muut lausekkeet voi rivittää ja formatoida mieleisekseen. Syntaksi esitetään käyttäen laajennettua BNF-notaatiota, joka noudattaa seuraavia merkintöjä:

- Terminaalit, eli tekstialkiot, on kirjoitettu lainausmerkkien "" sisään.
- Vaihtoehtoisuus ilmaistaan |-merkillä.
- Toisto kerran tai useammin ilmaistaan +-merkillä.
- Lisärajoitukset ilmaistaan #-merkillä alkavalla kommenttirivillä.

Molemmissa tehtävätyypeissä käytetään numeerisille lausekkeille samaa syntaksia, joka on esitetty kuvassa 3.1. Se suunniteltiin muistuttamaan suosittujen matemaattisten ohjelmien, kuten Matlab ja Maple syntaksia, jotta sen omaksuminen olisi helpompaa ja sen opettelusta olisi lisähyötyä. Tästä poikkeuksena syntaksi sallii kertolaskumerkin jättämisen pois sekä plus- ja miinusmerkkejä voi olla mielivaltainen määrä peräkkäin. Nämä muutokset tehtiin, jotta se olisi lähempänä paperilla laskemista. Numeeristen lausekkeiden jäsentämisen toteutus pohjautuu hieman eri muodossa olevaan BNF-kielioppiin, joka esitellään aliluvussa 5.3.

```
numlaus  ::= numlaus1 | numlaus "+" numlaus1 | numlaus "-" numlaus1
numlaus1 ::= numlaus2 | numlaus1 "*" numlaus2 | numlaus1 numlaus2
                      | numlaus1 "/" numlaus2
# Kolmas vaihtoehto yllä olevassa säännössä pätee vain kun numlaus2
# alkaa luvulla, muuttujalla tai sululla ja ei päde, jos numlaus1
# loppuu ja numlaus1 alkaa luvulla.
numlaus2 ::= numlaus3 | "+" numlaus2 | "-" numlaus2
numlaus3 ::= numlaus4 | numlaus4 "^" numlaus2
numlaus4 ::= numlaus5 | "(" numlaus ")" | "|" numlaus "|"
numlaus5 ::= luku | muuttuja
```

Kuva 3.1: Numeeristen lausekkeiden syntaksi.

Numeeristen lausekkeiden kieliopin aloitussymboli on `numlaus`, joka käsittää yhteen- ja vähennyslaskun. Välisymboli `numlaus1` vastaa puolestaan kerto- ja jakolaskua. Välisymboli sisältää myös implisiittisen kertolaskun, jota sen alapuolelta löytyvä lisärajoitus koskee. Hieman tulkinnanvaraista on, että sallitaanko implisiittiset kertolaskut, joissa jälkimmäinen tekijä on luku, eli esimerkiksi `(x+1)2`. Paperilla laskettaessa $2(x+1)$ tapaiset lausekkeet ovat yleisiä ja, vaikka $(x+1)2$ on harvinaisempi merkintä, sitäkin käytetään joskus, joten se sallitaan tässä kieliopissa. Esimerkiksi usean tekijän tulossa, jossa yksi tekijä supistuu luvuksi, on selkeätä pitää luku samassa paikkaa missä tekijä oli ainakin yhden ratkaisuaskeleen verran. Jakolaskuna joudutaan käyttämään `/`-merkkiä, jonka käyttö valitettavasti poikkeaa merkittävästi paperilla paljon käytettävästä jakoviivasta.

Välisymboli `numlaus2` mahdollistaa plus- ja miinusmerkkien käytön unaarioperaattoreina. Miinusmerkkejä saatetaan laittaa paperilla laskettaessa useita peräkkäin, mutta plusmerkillä sitä ei juuri tehdä. Koska merkintätavassa ei sinänsä ole mitään virheellistä, päätettiin se sallia ohjelman syntaksissa, vaikka sitä tuskin käytetään.

Välisymboli `numlaus3` vastaa potenssilaskua. Huomionarvoista on, että potenssilaskun oikea operandi on välisymboli `numlaus2`. Paperilla eksponentin erottaminen muusta lausekkeesta ei ole ongelma yläindeksimerkinnän ansiosta, mutta tekstipohjaisissa lausekkeissa asiaan pitää kiinnittää erityistä huomiota. Potenssimerkin `^` jälkeen tulevat etumerkit tulkitaan eksponenttiin kuuluviksi ilman sulkuja, koska x^{-2} tapaiset lausekkeet ovat yleisiä eikä niille ole muuta järkevää tulkintaa. Potenssimerkin jälkeen tulevat uudet potenssilaskut tulkitaan myös automaattisesti eksponenttiin kuuluviksi, eli `x^y^z` on sama kuin `x^(y^z)`. Tämä tulkinta on johdonmukainen muiden matemaattisten ohjelmien kanssa ja on myös intuitiivinen siinä mielessä, että paperillakin laskettaessa x^{y^z} tulkitaan $x^{(y^z)}$. Monimutkaisemmat lausekkeet eksponentissa vaativat sulkujen käyttöä, koska ne ovat harvinaisempia lukiotason

matematiikassa. Jos esimerkiksi kokonainen tulo tulkittaisiin automaattisesti eksponenttiin kuuluvaksi, lauseke x^2y täytyisi aina kirjoittaa muodossa `(x^2)y`, mikä olisi hyvin hankalaa.

Kieliopin loppupäässä välisymboli `numlaus4` vastaa sulkuja ja itseisarvoa ja välisymboli `numlaus5` vastaa muuttujaa ja lukua. Muuttuja on käytännössä yksi iso tai pieni kirjain `a` ja `z` väliltä aakkosellisessa järjestyksessä. Luku on puolestaan kokonaisluku, jossa sallitaan etunollat. Vaikka muuttuja ja luku ovat muuttuvia syntaksin osia, niihin viitataan jatkossa tekstialkioina aivan kuten operaattoreihinkin.

3.3 Laskutehtävän tarkastus

Laskutehtävien vastausten syntaksi on hyvin yksinkertainen. Se muodostuu kuvassa 3.1 esitetyistä numeerisista lausekkeista erotettuna yhtäsuuruusmerkein. Määritellyissä tehtävissä vastaus voi muodostua vain yhdestä numeerisesta lausekkeesta mutta vapaissa tehtävissä pitää olla ainakin yksi yhtäsuuruus. Muutenhan ohjelmalla ei ole mitään mitä tarkastaa.

Sekä määritellyissä että vapaissa laskutehtävissä vastauksessa esiintyvät yhtäsuuruudet tarkastetaan järjestyksessä alusta loppuun. Ensimmäiseksi yhtäsuuruuden tarkastuksessa jokaiselle muuttujalle määrätään n testiarvoa niiden arvoalueilta. Jos arvoalue muodostuu vähemmästä kuin n pisteestä, valitaan niin monta pistettä kuin mahdollista. Sitten käydään näiden testiarvojen jokainen kombinaatio läpi, evaluoidaan lausekkeiden arvot ja verrataan niitä keskenään. Jos löytyy yksikin testiarvojen kombinaatio, jolla lausekkeiden arvot eivät ole samat, ei yhtäsuuruus päde. Muuttujien ja lausekkeiden arvot, joilla yhtäsuuruus ei päde, ilmoitetaan käyttäjälle helpottamaan virheen selvittämistä.

Yhtäsuuruuksien tarkastusta voisi periaatteessa optimoida kokonaisuutena. Siinähän lasketaan kaikkien muiden paitsi ensimmäisen ja viimeisen lausekkeen arvot kahteen kertaan. Periaatteessa tarkastuksen voisi tehdä yksi testiarvojen kombinaatio kerrallaan. Kaikkien lausekkeiden arvot voisi laskea kerralla yhdellä testiarvojen kombinaatiolla ja sitten tarkastaa, että arvot ovat samat. Näin jokaisen lausekkeen arvo lasketaan vain kerran yhdellä kombinaatiolla. Tässä on kuitenkin ongelmana vaatimus, jonka mukaan vastauksen virheistä pitää ilmoittaa järjestyksessä. Kaikki kombinaatiot täytyisi siis käydä läpi pitäen kirjaa virheestä lähinnä vastauksen alkupäätä. Se saattaa olla hitaampi operaatio kuin yhtäsuuruuksien läpikäyminen järjestyksessä, jos ensimmäinen virhe on lähellä vastauksen alkupäätä. Tämä menettely ei myöskään sovellu yhtälönratkaisutehtäviin, joita tarkastellaan aliluvussa 3.4. Ohjelman toteutuksen kannalta olisi tietenkin hyvä, jos samaa menetelmää voisi käyttää molemmissa tehtävätyypeissä. Tällä optimoinnilla saavutettua parannusta ei katsottu tarpeeksi suureksi hyödyksi verrattuna toteutuksen vaikeuksiin.

Toinen vaihtoehto on tallettaa lausekkeiden arvot kaikille testiarvojen kombinaa-

tioille. Tällöin operaation muistinkulutus olisi tosin $\theta(s^n)$, missä s on muuttujien lukumäärä. Tämäkään vaihtoehto ei siten ole selvästi parempi, joten ohjelma käyttää ensimmäisenä kuvattua menetelmää.

Yhtäsuuruuden tarkastus arvoja kokeilemalla on itse asiassa varma muutamassa rajoitetussa tapauksessa olettaen, että pysytään ohjelman määräämän lukualueen sisällä ja mahdolliset pyöristysvirheet laskuissa eivät kasva liian suuriksi. Yksinkertaisimmassa tapauksessa ei ole ollenkaan muuttujia, jolloin ohjelman on helppo tarkastaa, että laskut pitävät paikkansa. Lukualueen rajallisuuteen ja laskuoperaatioiden mahdollisiin pyöristysvirheisiin palataan aliluvussa 5.2.

Tarkastellaan yhden muuttujan x polynomilausekkeiden $p_1(x)$ ja $p_2(x)$ yhtäsuuruuden tarkastusta, kun ne ovat enintään astetta m. On itsestään selvää, että jos yksikin n:stä testiarvosta johtaa eri arvoihin lausekkeiden evaluoinnissa, ei yhtäsuuruus päde. Oletetaan siis, että erisuuruisilla testiarvoilla $x_1, x_2, \ldots, x_n$ pätee

$$p_1(x_i) = p_2(x_i), \text{kun } 1 \leq i \leq n.$$

Havaitaan, että

$$p_1(x) = p_2(x) \iff p_1(x) - p_2(x) = 0,$$

jossa polynomi $p_1(x) - p_2(x) = p(x)$ on enintään astetta m. Havaitaan myös, että polynomilla $p(x)$ täytyy olla ainakin n nollakohtaa, nimittäin $x_1, x_2, \ldots, x_n$. Polynomilaskennan perustuloksen mukaan m asteen polynomilla voi olla enintään m nollakohtaa. Jos $m < n$, $p(x)$ ei voi olla $n-1$ asteen tai pienemmänkään asteen polynomi, jolloin sen täytyy olla nollafunktio. Näin ollen, kun $m < n$ niin $p(x) = 0$ ja $p_1(x) = p_2(x)$. Toisin sanoen, kun käytetään n testiarvoa, enintään astetta $n-1$ olevien yhden muuttujan polynomilausekkeiden tarkastus on varma.

Edellä esitetty todistus ei oleta muuttujan testiarvoilta mitään muuta kuin, että ne ovat erisuuruisia. Tämän takia muuttujien testiarvot valitaan lähinnä sillä perusteella, että ne olisivat käyttäjälle helpot arvot virheen selvityksessä, jos sellainen löytyy. Ensinnäkin valitaan arvo 0, jos se kuuluu muuttujan arvoalueeseen. Seuraavaksi valitaan arvoja, jotka ovat arvoalueen äärellisillä rajoilla. Sitten valitaan kokonaislukuja suosien sellaisia, jotka ovat lähellä arvoalueen äärellisiä rajoja. Jos kokonaislukuja ei löydy tarpeeksi, etsitään samalla tapaa lukuja puolikkaan välein, sitten neljäsosan välein ja niin edelleen.

Tällaiset tehtävät kattavat jo merkittävän osan kirjan [5] laskutehtävistä. Ohjelmassa n arvoksi valittiin seitsemän, koska se riittää kirjan tehtäviin ja suurempi arvo hidastaisi ohjelman toimintaa turhaan. Korkeampiasteisia polynomeja sisältävien tehtävien käyttö olisi tuskin pedagogisestikaan hyväksi. Kirjoittajan tiedossa ei ole, pystytäänkö vastaavia tuloksia johtamaan monimutkaisemmille tehtäville. Käytännön kokeilut osoittavat kuitenkin, että monet virheet löytyvät monimutkaisem-

mistakin lausekkeista.

Määritellyissä laskutehtävissä tehdään vastauksen yhtäsuuruuksien tarkastuksen lisäksi muita tarkastuksia. Ensinnäkin tarkastetaan vastauksen ensimmäisen lausekkeen yhtäsuuruus määritellyn lopullisen lausekkeen kanssa. Yhdessä nämä tarkastukset takaavat, että vastauksen kaikki lausekkeet ovat yhtä suuria lopullisen lausekkeen kanssa olettaen, että yksittäisten yhtäsuuruuksien tarkastukset ovat varmoja. Tarkastus suoritetaan lopullisen lausekkeen eikä aloituslausekkeen kanssa, koska aloituslauseketta ei välttämättä ole edes määritelty. Lisäksi aloituslauseke on tehtävän määrittelytiedoston luvun yhteydessä tarkastettu yhtä suureksi lopullisen lausekkeen kanssa.

Lopuksi tarkastetaan onko viimeinen lauseke sopivassa muodossa. Oletuksena katsotaan onko viimeisessä lausekkeessa lukujen, muuttujien esiintymien ja operaattoreiden yhteenlaskettu lukumäärä pienempi tai yhtä suuri kuin lopullisessa lausekkeessa. Implisiittiset kertolaskut lasketaan operaattoreiksi ja sulkuja ei lasketa ollenkaan mukaan. Näin pyritään antamaan käyttäjälle vapaus käyttää mieleistään merkintätapaa. Lisäksi itseisarvo lasketaan yhdeksi operaattoriksi, vaikka koostuu kahdesta tekstialkiosta. Lopullinen lauseke antaa siis vain esimerkin kuinka yksinkertaiseen muotoon lauseke pitää vähintään saada. Tämä menettely toimii monissa tehtävissä, koska usein lopullinen lauseke on yksinkertaisempi kuin mistä lähdetään liikkeelle. Yleensä ei haluta vaatia lopullisen lausekkeen kanssa identtistä viimeistä lauseketta, koska lausekkeet voi kirjoittaa monella tavalla. Esimerkiksi summan termien ja kertolaskun tekijöiden järjestystä voi vaihtaa muuttamatta lausekkeen merkitystä. Olisi toki mahdollista toteuttaa ohjelmaan lausekkeiden vertailu, joka ottaa tällaisia muunnoksia huomioon. Sellaista ominaisuutta ei toteutettu, koska se ei ole ohjelman toiminnan kannalta ensisijaisen tärkeää.

Tiettyjä laskutehtäviä varten ohjelmassa on kuitenkin mahdollisuus vaatia, että viimeinen lauseke on identtinen lopullisen lausekkeen kanssa. Tässä tarkastuksessa implisiittinen ja eksplisiittinen kertolasku tulkitaan samaksi operaatioksi ja sulut otetaan huomioon. Tätä vaihtoehtoa tarvitaan esimerkiksi sellaisissa tehtävissä, joissa lopullinen lauseke on pidempi kuin aloituslauseke. Näissä tehtävissä jää tehtävän laatijan vastuulle antaa sanallisessa ohjeessa tarpeeksi tarkat ohjeet käyttäjälle viimeisen lausekkeen oikeasta muodosta.

Laskutehtävien määrittelytiedostojen formaatti esitetään kuvassa 3.2 esimerkin avulla. Tiedostoformaatin perusrakenne on `avainsana="arvo"`. Tehtävien määrittelytiedostojen avainsanat ovat englannin kielellä, syistä jotka kerrotaan luvussa 4. Tehtävän muuttujat ja niiden arvoalueet määritellään avainsanalla `symbolRanges`. Eri muuttujien määrittelyt erotellaan pilkulla ja itse määrittelyt noudattavat samaa syntaksia kuin yhtälönratkaisutehtävien lukuvälilausekkeet, jotka esitellään aliluvussa 3.4. Syntaksin tiivis kuvaus löytyy kuvasta 3.5. Aloituslauseke ja lopullinen lause-

```
exerciseType="calculation"
symbolRanges="x, a /= 0"
startExpression="(x + a^2)/a - (a - x/a)"
finalExpression="2x/a"
requireExactMatch="false"
explanation="Laske lausekkeen arvo."
```

Kuva 3.2: Esimerkki laskutehtävän määrittelytiedostosta.

```
looglaus      ::= looglaus1 | looglaus "tai" looglaus1
looglaus1     ::= looglaus2 | looglaus1 "ja" looglaus2
looglaus2     ::= vertailuKetju | "(" looglaus ")"
vertailuKetju ::= numlaus (("="|"/="|"<"|"<="|">="|">") numlaus)+
```

Kuva 3.3: Loogisten lausekkeiden syntaksi.

ke noudattavat samaa syntaksia kuin vastauksen numeeriset lausekkeet kuvassa 3.1. Jos ei haluta antaa aloituslauseketta, voidaan sen arvo jättää tyhjäksi merkkijonoksi tai jättää koko avainsana pois. Avainsanan `requireExactMatch` arvoilla `false` ja `true` määritellään täytyykö viimeisen lausekkeen olla identtinen lopullisen lausekkeen kanssa. Avainsanalla `explanation` määritellään tehtävänannon tekstuaalinen osa.

3.4 Yhtälönratkaisutehtävän tarkastus

Yhtälönratkaisutehtävissä käytetään numeeristen lausekkeiden sijasta loogisia lausekkeita. Käytännössä tämä tarkoittaa vain syntaksin laajentamista siten, että se mahdollistaa numeeristen lausekkeiden vertailun keskenään ja tarjoaa loogisten operaattoreiden käytön vertailujen kesken. Loogisten lausekkeiden syntaksi on esitetty kuvassa 3.3. Siinä esiintyvä välisymboli `numlaus` viittaa numeeriseen lausekkeeseen eli kuvan 3.1 vastaavaan välisymboliin. Syntaksi sallii vertailuoperaattoreiden ketjuttamisen, joka on yleistä paperilla laskemisessa ja säästää kirjoittamista. Esimerkiksi lauseke `1 < x < 2` tulkitaan $1 < x \wedge x < 2$. Erityismaininnan vaatii erisuuruusoperaattorin ketjuttaminen, jota saattaa helposti käyttää väärin. Lauseke `x /= y /= z` tarkoittaa $x \neq y \wedge y \neq z$, joka puolestaan ei takaa, että $x \neq z$. Tämä on tietenkin suora seuraus siitä, että erisuuruus ei ole transitiivinen relaatio. Loogisten lausekkeiden jäsentämisen toteutus pohjautuu hieman eri muodossa olevaan BNF-kielioppiin, joka esitellään aliluvussa 5.3.

Aliluvussa 2.3 kuvatun haarautumisrakenteen ohjelma toteuttaa ehtolohkoiksi nimetyllä rakenteella. Ehtolohkojen syntaksi esitetään kuvissa 3.4 ja 3.5. Lukuvälien

```
ehtoLohko    ::= looglausKetju | haarauma
                               | looglausKetju "<==>" haarauma
looglausKetju ::= looglaus | "<==>" looglausKetju
haarauma     ::= haara+
haara        ::= "kun" ehto "niin" ehtoLohko ";"
ehto         ::= lukuVälit
# Saman haarauman ehtojen täytyy koskea samaa muuttujaa.
```

Kuva 3.4: Yhtälönratkaisutehtävän ehtolohkojen syntaksi.

syntaksi esitetään erillisessä kuvassa, koska samaa syntaksia käytetään tehtävien määrittelytiedostoissa muuttujien arvoalueiden määrittelemiseen. Ehtolohkot muodostavat yhtälönratkaisutehtävien syntaksin korkeimman tason. Toisin sanoen yhtälönratkaisutehtävän vastaus on kokonaisuudessaan yksi ehtolohko. Niissä on kaksi vaihtoehtoista osaa: lausekeketju ja haarauma. Ehtolohko voi muodostua molemmista tai vain toisesta osasta. Lausekeketju on ehtolohkon alkuosa, joka on loogisten lausekkeiden yhtäpitävyyksistä muodostuva päättelyketju. Haaraumassa päättelyketju jakautuu haaroihin ehtojen mukaan. Haaraumakohdan yhtäpitävyyden täytyy tietenkin päteä kaikkien haarojen ensimmäisten lausekkeiden kanssa. Sisäkkäisten ehtolohkojen ehdoissa ei tarvitse toistaa ulompien ehtoja. Esimerkiksi, jos halutaan sisempi ehtolohko, jonka haarat ovat lukuväleillä $[-1, 0)$ ja $[0, 1]$, kun ulomman ehtolohkon haaran ehto on $-1 \leq x \leq 1$, sisemmän ehtolohkon haarat voidaan kirjoittaa $x < 0$ ja $x \geq 0$. Täsmällisemmän ehdon kirjoittamisesta ei tietenkään ole haittaa, mutta ohjelma noudattaa tätä periaatetta, koska paperille kirjoitettaessa tulkinta olisi yleensä sama.

Huomionarvoista on, että ehtolohkon haaroja ei voi yhdistää. Vastaus on siis puumainen rakenne, jossa päättelyketju vain haaroittuu. Yhdistämisen puuttuminen ei rajoita tehtävien kattavuutta, koska vastauksen, joka käyttäisi haarojen yhdistämistä, voisi kirjoittaa ilman yhdistämistä toistamalla yhdistämisen jälkeisen päättelyketjun kaikissa haaroissa. Käytännössä haarojen yhdistämisen salliminen säästäisi käyttäjältä kirjoittamista ja johtaisi selkeämpiin vastauksiin. Sen toteuttaminen todettiin kuitenkin liian hankalaksi tämän projektin aikataulun puitteissa verrattuna siitä saatuun hyötyyn.

Kuten kuvasta 3.5 nähdään, lukuvälien syntaksi on laadittu vastaamaan loogisten lausekkeiden syntaksia. Käytettävyyden parantamisen lisäksi se on johdonmukainen haarautumisrakenteen perusidean kanssa. Paperilla laskettaessahan ehdon paikalla voisi periaatteessa käyttää mitä tahansa loogista lauseketta. Tämä logiikan merkintöihin perustuva syntaksi voidaan helposti tulkita lukuvälien määrittelyksi.

```
lukuVälit ::= jaVälit | jaVälit "tai" lukuVälit
jaVälit   ::= väli0 | väli0 "ja" jaVälit
väli0     ::= väli1 | väli2 | väli3
väli1     ::= muuttuja ("="|"/="|"<"|"<="|">="|">") numlaus
väli2     ::= numlaus ("="|"/="|"<"|"<="|">="|">") muuttuja
väli3     ::= numlaus ("<"|"<=") muuttuja ("<"|"<=") numlaus
# Välisymbolit numlaus eivät saa sisältää muuttujia.
```

Kuva 3.5: Lukuvälien määrittelyssä käytettävä syntaksi.

Välisymboleilla `väli1`, `väli2` ja `väli3` määritellään lukusuoran osia, joita voi yhdistellä leikkaus- ja unionioperaatioilla käyttäen merkintöjä `ja` ja `tai` vastaavasti. Välisymbolilla `väli1` voidaan määritellä yksittäinen piste tai yhtenäinen lukusuoran osa, joka jatkuu äärettömiin toisesta päästä. Merkinnällä `/=` voidaan itse asiassa määritellä kaksi yhtenäistä väliä, jotka molemmat jatkuvat äärettömyyteen ja joita erottaa vain yksi piste. Välisymbolilla `väli2` voi määritellä täysin samat lukuvälit mutta toisin päin kirjoitettuna. Välisymbolilla `väli3` puolestaan määritellään yhtenäinen lukusuoran väli, jonka molemmat päät on rajattu. Kaikissa väleissä äärellinen raja voi kuulua tai olla kuulumatta väliin.

Kuten edellisessä kappaleessa mainittiin, ohjelman täytyy tarkastaa, että ehtolohkon haarat kattavat kaikki mahdollisuudet. Lähinnä tämän takia yleinen ehto on korvattu lukuvälin määrittelyllä ja yhdessä ehtolohkossa sallitaan vain yhtä muuttujaa koskevia ehtoja. Näillä rajoituksilla tarkastus voidaan tehdä suhteellisen pienellä vaivalla, kunhan käytettävissä on lukuvälien unioni-, leikkaus- ja vertailuoperaatiot. Tarkastus voidaan tehdä ehtolohko kerrallaan. Otetaan jokaisen haaran lukuväli, muodostetaan niiden leikkaukset haarauman muuttujan arvoalueen kanssa ja muodostetaan niiden unioni. Jos se täsmää muuttujan arvoalueen kanssa, kattavat ehtolohkon haarat koko arvoalueen. Tämä toimenpide toistetaan sitten joka haaran ehtolohkolle ottaen huomioon haarauman muuttujan uusi arvoalue. Haarojen lukuvälien sallitaan menevän päällekkäin, koska vastaus saattaa olla täysin oikein vaikka näin on.

Jos samassa ehtolohkossa sallittaisiin usean muuttujan käyttö esimerkiksi sallimalla `ja`- ja `tai`-operaattorien käyttö eri muuttujien välien kesken, täytyisi kattavuuden tarkastuksessa käsitellä muuttujien arvoalueiden kombinaatioita. Toisin sanoen täytyisi tutkia n-ulotteisen avaruuden osien leikkauksia ja unioneita, missä n on tehtävän muuttujien lukumäärä. Esimerkiksi ehto $x > 0 \wedge y > 0$ määrittelee yhden tason neljänneksen ja ehto $x > 0 \wedge y > 0 \wedge z > 0$ määrittelee kahdeksasosan kolmiulotteisesta avaruudesta. Usean muuttujan käytön samassa ehtolohkossa voi myös kiertää muilla ohjelman ominaisuuksilla. Ehtolohkoja voi määritellä sisäkkäin,

mikä korvaa `ja`-operaattorien käytön. Operaattori `tai` voidaan korvata tekemällä sisäkkäisiä ehtolohkoja ja toistamalla sama päättelyketju useassa haarassa. Esimerkiksi kahta muuttujaa sisältävää ehtolohkoa

```
kun x = 0 ja y = 0 niin L1;
kun x = 1 tai y = 1 niin L2;
kun x /= 0 ja x /= 1 tai y /= 0 ja y /= 1 niin L3;
```

vastaa yhden muuttujan sisäkkäiset ehtolohkot

```
kun x = 0 niin
  kun y = 0 niin L1;
  kun y = 1 niin L2;
  kun y /= 0 ja y /= 1 niin L3;
kun x = 1 niin L2;
kun x /= 0 ja x /= 1 niin
  kun y = 1 niin L2;
  kun y /= 1 niin L3;
```

Ei ole aivan triviaalia toteuttaa n-ulotteisten välien leikkausta, unionia sekä vertailua ja toisaalta rajoittuminen yhteen muuttujaan ei merkittävästi rajoita käyttäjää, joten yhden muuttujan säännössä pitäytyminen nähtiin parhaaksi ratkaisuksi.

Jos sallittaisiin minkä tahansa loogisen lausekkeen käyttö ehtona, olisi haarojen kattavuuden tarkastuksen toteuttaminen erittäin vaikeaa. Ensinnäkin ehdoissa voisi taas esiintyä useita eri muuttujia. Toiseksi ohjelman täytyisi pystyä päättelemään loogisesta lausekkeesta sen asettamat rajoitukset muuttujille. Esimerkiksi, jos ehto olisi muotoa $ax^2+bx+c = 0$ täytyisi ohjelman osata ratkaista toisen asteen yhtälöitä. Yleisemmin ehto voisi olla muotoa $f(x) = 0$, missä $f(x)$ voi olla mikä tahansa ohjelman operaattoreilla muodostettavissa oleva funktio, jonka nollakohdat pitäisi ratkaista. Tämä on mahdotonta ilman numeerisia menetelmiä ja ei ole tietenkään toteutettavissa tämän diplomityön puitteissa.

Yhtälönratkaisutehtävän yhtäpitävyyksien tarkastus suoritetaan samalla menetelmällä kuin yhtäsuuruuksien tarkastus laskutehtävissä. Kuten aliluvussa 3.3 mainittiin, menetelmän optimointi on vielä vaikeampaa yhtälönratkaisutehtävissä. Ehtolohkot aiheuttavat sen, että samoja muuttujien testiarvoja ei aina voida käyttää kaikkien yhtäpitävyyksien tarkastuksessa. Tästäkin syystä ohjelma tarkastaa yhtäpitävyydet järjestyksessä molemmissa tehtävätyypeissä.

Muuttujien testiarvojen valinta on paljon kriittisempi osa tarkastusta kuin laskutehtävissä. Pienikin virhe numeerisessa lausekkeessa yleensä muuttaa sen saamia arvoja kaikilla muuttujien arvoilla huomattavasti, joten testiarvon valintaan ei tarvitse kiinnittää paljon huomiota laskutehtävissä. Yhtälönratkaisutehtävissä puolestaan käsitellään loogisia lausekkeita, joiden evaluoinnin tuloksena voi olla vain kaksi

eri arvoa. Jos testiarvot valittaisiin samalla tavalla kuin laskutehtävissä, on paljon todennäköisempää, että lausekkeet saavat saman totuusarvon vaikka lausekkeet eivät ole yhtäpitäviä. Lisäksi, jos loogisten lausekkeiden vertailut ovat yhtäsuuruuksia, on erittäin epätodennäköistä, että satutaan valitsemaan kertaakaan muuttujien arvot joilla yhtäsuuruus pätee.

Tästä syystä yhtälönratkaisutehtävissä ohjelmaan tarvitaan toiminnallisuutta, joka valitsee sopivia testiarvoja aliluvussa 3.3 kuvatun muuttujien arvoalueiden rajoja painottavan valinnan lisäksi. Käytännössä nämä testiarvot poimitaan vastauksen vertailuista ja määritellyissä tehtävissä myös oikean vastauksen vertailuista. Jos vertailulauseke on sellaisessa muodossa, että toisella puolella on yksin muuttuja ja sama muuttuja ei esiinny toisella puolella, voidaan siitä poimia testiarvoja. Jos vertailun toisella puolella ei esiinny muuttujia, yksinkertaisesti lasketaan sen lausekkeen arvo y ja käytetään sitä testiarvona. Jos vertailu on $=$ tai $\neq$, tämä testiarvo takaa, että vertailu saa ainakin kerran arvon tosi tai vastaavasti arvon epätosi. Vertailu saa hyvin todennäköisesti myös toisen arvonsa, kun valitaan loppuja testiarvoja. Jos vertailu on $<$ tai $\geq$, otetaan myös arvo $y - \epsilon$, missä ϵ on pieni positiivinen luku. Vastaavasti, jos vertailu on $>$ tai $\leq$, otetaan myös arvo $y + \epsilon$. Nämä takaavat, että loputkin vertailut saavat molemmat arvonsa.

Testiarvojen valinta on monimutkaisempaa, jos vertailun toisellakin puolella on muuttujia. Oletetaan, että vertailun toisella puolella on yksin muuttuja x. Tällöin ohjelma kokeilee toisen puolen muuttujille arvoja ja laskee arvon y sen puolen lausekkeelle. Tämän jälkeen pitää vielä tarkastaa osuuko arvo muuttujan x arvoalueelle. Jos osuu, lisätään muuttujalle x testiarvo y sekä muille muuttujille löydetyt testiarvot. Muuttujalle x lisätään myös vertailusta riippuen muita testiarvoja samalla tavalla kuin edellisessä kappaleessa kuvataan. Tässä menettelyssä on mahdollista, että ei löydetä testiarvoja, jotka sopivat muuttujien arvoalueisiin. Käytännössä tämä ei kuitenkaan muodostunut ongelmaksi, koska useimmissa tehtävissä muuttujilla on tarpeeksi suuret arvoalueet ja ongelma koskee vain usean muuttujan tehtäviä.

Sekä vapaissa että määritellyissä tehtävissä testiarvot kerätään koko vastauksesta ennen yhtäpitävyyksien tarkastusta. Niistä käytetään mahdollisimman monia kaikissa tarkastuksissa, kunhan testiarvot ovat muuttujien arvoalueiden sisällä. Yhdessä tarkastuksessa käytettävien testiarvojen kokonaismäärää on rajoitettu seitsemään muuttujaa kohti. Edellä kuvattulla tavalla valittujen testiarvojen lisäksi loput testiarvot valitaan samalla menetelmällä kuin laskutehtävissä, eli painottaen muuttujien arvoalueiden rajoja, ja välttäen jo valittuja testiarvoja.

Seitsemän testiarvoa osoittautui riittäväksi testauksessa käytetyille kirjan [5] tehtäville. Suurempi raja vain hidastaisi ohjelmaa, joten rajaa ei asetettu korkeammalle. Varsinaisissa yhtälönratkaisutehtävissä, oli enimmillään viisi ratkaisua, joten viiden erikseen poimitun testiarvon lisäksi jäi yli kaksi testiarvoa, joilla yhtälö ei voi päteä.

Epäyhtälöiden ratkaisut puolestaan muodostuivat maksimissaan kahdesta vertailusta, joista tulee siis yhteensä neljä testiarvoa, jolloin kolme testiarvoa jää vielä yli. Molemmissa tapauksissa siis jää testiarvoja yli siten, että muuttujien arvoalueiden rajojakin testataan. Testiarvojen lukumäärää rajoitetaan, jotta tarkastus ei veisi liian kauan aikaa. Jos ohjelman tulee selvitä monimutkaisemmista yhtälönratkaisutehtävistä, täytyy rajaa nostaa.

Vapaissa tehtävissä testiarvoja voidaan etsiä vain itse vastauksesta. Tyypillisessä tapauksessa vastauksen vertailut, joista löydetään testiarvoja, ovat vastauksen lopussa, eli käyttäjän saama ratkaisu. Käytännössä ohjelma siis tarkastaa, että vastauksen päättelyketjut pätevät käyttäjän saamalla ratkaisulla ja testaa muutamaa muuta testiarvoa. On kuitenkin hyvin epätodennäköistä, että ohjelma huomaisi ratkaisusta puuttuvan osan. Esimerkiksi, jos tehtävä on ratkaista yhtälö $x^2 = 100$ ja käyttäjän vastaus on $x = 10$, ei ohjelma osaa testata arvolla $x = -10$ eikä ohjelma löydä virhettä.

Määritellyissä tehtävissä ohjelma löytää myös virheet, joissa ratkaisusta puuttuu osa, koska se on saanut testiarvot myös määrittelytiedoston oikeasta vastauksesta. Näin ollen ero vapaiden ja määriteltyjen tehtävien tarkastuksen varmuudessa on paljon suurempi yhtälönratkaisutehtävissä kuin laskutehtävissä.

Vapaissa tehtävissä ainoat tarkastukset, jotka ohjelma tekee ovat vastauksessa esiintyvien yhtäpitävyyksien tarkastus ja ehtolohkojen kattavuuden tarkastus. Määritellyissä tehtävissä ohjelma tekee lisäksi muita tarkastuksia. Ensinnäkin tarkastetaan, että vastauksen ensimmäiset lausekkeet ovat yhtäpitäviä oikean vastauksen kanssa. Nämä tarkastukset takaavat, että koko vastaus on yhtäpitävä oikean vastauksen kanssa olettaen, että yksittäisten yhtäpitävyyksien tarkastukset ovat varmoja. On syytä huomauttaa, että oikea vastaus on yleisesti ottaen ehtolohko, jotta vastauksena voi olla haaroittunut rakenne. Tämän takia joudutaan toteuttamaan yhtäpitävyyden tarkastus myös ehtolohkojen välillä, minkä ohjelma tekee tarkastamalla ehtolohkojen ensimmäisiä lausekkeita keskenään ottaen huomioon muuttujien arvoalueet. Jos oikeassa vastauksessa ei tarvita haaroja, sitä esittävä looginen lauseke on ehtolohko, jolla ei ole haaraumaa ja jonka lausekeketju muodostuu yhdestä lausekkeesta.

Lopuksi tehdään vastauksen loppupäälle vielä kolme tarkastusta, jotka tutkivat onko vastaus sievennetty tarpeeksi pitkälle. Ensin tarkastetaan, että kaikki viimeiset loogiset lausekkeet ovat ratkaistussa muodossa, eli ratkaistava muuttuja esiintyy vain vertailujen toisella puolella yksin. Toiseksi tarkastetaan, että vastauksen haarojen lukumäärä on enintään oikean vastauksen haarojen lukumäärä. Lopuksi verrataan viimeisten lausekkeiden lukujen, muuttujien esiintymien ja operaattoreiden yhteistä lukumäärää oikean vastauksen vastaavaan lukuun samoin säännöin kuin laskutehtävissä.

```
symbolRanges="x /= 0, y > 0"
symbolToSolve="x"
presetTestValues="x = 1, -1, 2, -2; y = 1, -1, 2, -2;"
equation="y/x = x"
solution="x = y^(1/2) tai x = -y^(1/2)"
explanation="Ratkaise yhtälö muuttujan x suhteen."
```

Kuva 3.6: Esimerkki yhtälönratkaisutehtävän määrittelytiedostosta.

Esimerkki yhtälönratkaisutehtävän määrittelytiedostosta esitetään kuvassa 3.6. Tehtävän muuttujat ja niiden arvoalueet määritellään täsmälleen samalla tavalla kuin laskutehtävissä. Avainsanalla `symbolToSolve` määritellään ratkaistava muuttuja. Koska testiarvojen valinta on kriittistä yhtälönratkaisutehtävissä, on tehtävän laatijalle annettu mahdollisuus määrätä testiarvot avainsanalla `presetTestValues`, mutta avainsanan voi jättää myös pois. Itse arvot annetaan formaatissa, jossa muuttuja on ensimmäisenä, minkä jälkeen tulee yhtäsuuruusmerkki ja muuttujan arvot. Arvot erotetaan toisistaan pilkulla ja loppumerkkinä toimii puolipiste. Kaikille muuttujille voi antaa haluamansa määrän testiarvoja ja kaikkia muuttujia ei ole edes pakko listata. Määrittelytiedostossa annetut testiarvot menevät kaikkien muiden edelle testiarvojen valinnassa. Aloitusyhtälö määritellään avainsanalla `equation` ja oikea ratkaisu avainsanalla `solution`. Ne noudattavat ehtolohkojen syntaksia kuvassa 3.4, mutta niissä ei saa esiintyä yhtäpitävyysmerkkejä. Tekstuaalinen ohje määritellään jälleen avainsanalla `explanation`.

4. YLEISTÄ TOTEUTUKSESTA

Ohjelman web-käyttöliittymän toteutuksessa päätettiin käyttää CGI-tekniikkaa [13]. CGI-tekniikan perusajatus on, että palvelupyynnön saadessaan web-palvelin käynnistää erillisen CGI-ohjelman, joka puolestaan tuottaa web-sivun. Web-palvelin myös välittää CGI-ohjelmalle selaimelta saamansa datan. CGI-tekniikka on siis pohjimmiltaan hyvin yksinkertainen tapa toteuttaa web-sovellus. Ohjelman käyttöliittymä on hyvin yksinkertainen, joten sen toteutus onnistuu pienellä vaivalla pelkästään CGI-tekniikalla. Tämän työn päämääränä on keskittyä varsinaisen tarkastuksen toteuttavaan ohjelman osaan, joten monimutkaisempien web-tekniikoiden käyttöä ei harkittu sen pidemmälle.

Ohjelman toteutuskielen valinta tehtiin lähinnä kahden kielen välillä: Python ja C++. Python soveltuu nopeaan sovellusten kehitykseen [14], mikä sopi tähän projektiin hyvin. Toinen Pythonin etu on, että se on suosittu kieli web-sovellusten toteutuksessa CGI-tekniikalla. Sen käytöstä CGI-tekniikan kanssa löytyy siis paljon materiaalia, mikä helpottaa web-puolen toteutusta merkittävästi.

Tulkattavana kielenä Pythonin huono puoli on tietenkin tehottomuus, kun taas C++ on hyvin tehokas korkean tason ohjelmointikieli. Vaikka tässä ohjelmassa onkin muutamia algoritmeja, joiden tehokkuus voi muodostua ongelmaksi, lähdettiin toteutuksessa liikkeelle Pythonilla. Pythonilla on sekin etu, että sitä voi laajentaa omalla C++-koodilla, eli nopeuskriittiset osat voisi tarvittaessa kirjoittaa uusiksi C++-kielellä. Toteutus pystyttiin kuitenkin viemään loppuun asti Pythonilla, mutta tehokkuuteen täytyi kiinnittää erityistä huomiota tietyissä ohjelman osissa.

Ohjelmakoodi päätettiin kirjoittaa englanniksi. Suomea ei käytetty, koska ohjelman jatkokehityksen haluttiin olevan mahdollista myös suomea taitamattomille. Lisäksi haluttiin, että ohjelmasta olisi helppo tehdä eri kieliversioita ja erityisesti englanninkielinen versio. Jos ohjelmaa testattaisiin TTY:n tapaisessa oppilaitoksessa, on hyvinkin mahdollista, että tarvitaan englanninkielinen versio. Jos monikielisyyden tukea ei ota huomioon ohjelman suunnittelussa, sen toteuttaminen jälkeenpäin on hyvin hankalaa. Vain web-käyttöliittymä on tarkoitus kääntää uuden kielen tukea toteutettaessa. Tehtävien laatijoille näkyvät tehtävien määrittelytiedostot ja ohjelman ylläpitäjälle näkyvä konfigurointitiedosto ovat aina englanninkielisiä, koska on kohtuullista olettaa, että tällaiset henkilöt osaavat muutenkin englantia.

Suuri osa tämän ohjelman toiminnallisuutta on vastausten jäsentäminen. Au-

tomaattisesti jäsentäjiä tuottavien ohjelmien, kuten YACC [15] käyttöä harkittiin, mutta lopulta jäsentäjät päädyttiin toteuttamaan itse. Suurin syy oli, että itse toteutetussa jäsentäjässä voi syntaksivirheiden käsittelyn tehdä miten haluaa, mikä helpottaa hyvien virheilmoitusten antamista. Hyvät virheilmoitukset syntaksivirheistä ovat kriittisiä tämän ohjelman käytettävyyden kannalta.

Ohjelman rakenteen ymmärtämiseksi täytyy tuntea muutamia Pythonin perusrakenteita. Python-ohjelmat muodostuvat korkealla tasolla *moduuleista* (eng. *module*) ja *paketeista* (eng. *package*). Moduulit ovat ohjelmakoodin perusyksiköitä, joihin kaikki koodi jaotellaan. Pakkaukset ovat puolestaan hierarkkinen rakenne, jolla moduulit voidaan järjestää. Näiden lisäksi ohjelma hyödyntää olio-ohjelmoinnin perusrakenteita, kuten luokat ja oliot sekä jossakin määrin periyttämistä.

Ohjelma jakautuu kahteen pääosaan. Ohjelman ytimen muodostavat moduulit, jotka suorittavat tehtävien tarkastuksen, paketissa nimeltä *backend*. Loput moduulit toteuttavat web-käyttöliittymän, käyttäen hyväkseen pakettia *backend*. Tarkastuksen eristäminen on luonnollinen tapa jäsentää ohjelma ja tekee sen uudelleenkäytöstä helpompaa. Paketin *backend* toteutus käydään läpi aliluvussa 5.1 ja web-käyttöliittymä esitellään luvussa 6.

Erityisen hankalaksi paketin *backend* ja käyttöliittymän rajapinnassa osoittautui tiedon välittäminen käyttäjän virheistä, jotka kuuluvat ohjelman normaaliin toimintaan. Tällaisia ovat lähinnä virheet vastauksen syntaksissa ja oikeellisuudessa. Virheitä on hyvin monenlaisia ja kaikissa niissä tulisi käyttäjälle välittää jotain informaatiota auttamaan virheen korjaamisessa. Ei ole järkevää määritellä ja muodostaa virheilmoituksia paketin *backend* koodissa, joten paketista *backend* palautetaan lista virhekoodeja ja muuta tarvittavaa tietoa hyvän virheilmoituksen antamiseksi. Virhekoodeja on lista, koska tietyn tyyppisiä virhekoodeja käytetään monessa yhteydessä, jolloin toisella virhekoodilla täsmennetään missä yhteydessä virhe sattui. Esimerkiksi syntaksivirheitä syntyy sekä vastauksen että määrittelytiedoston jäsentämisessä. Listan ensimmäinen virhekoodi kertoo mitä jäsentäessä syntaksivirhe sattui ja seuraavat koodit kertovat millaisesta syntaksivirheestä on kyse. Virhekoodien ohessa välitettävä tieto voi olla esimerkiksi syntaksivirheen paikka vastauksessa tai muuttujien arvot, joilla vastaus ei ole oikein. Johdonmukaisuuden takia samaa virherakennetta käytetään monien muidenkin ohjelman virheiden välityksessä.

Toinen mainitsemisen arvoinen asia paketin *backend* rajapinnassa on tarkastuksen suorittavien funktioiden aikakatkaisu, eli tarkastuksen keskeytys, kun se kestää liian kauan. Aikakatkaisu tarvitaan, koska vastauksen pituutta ei rajoiteta, jolloin tarpeeksi pitkän ja monimutkaisen vastauksen antamalla tarkastus voi kestää hyvin pitkään. Aikakatkaisu on siis toteutettu täysin paketissa *backend* ja aikakatkaisun tapahtuessa tarkastusfunktiot yksinkertaisesti palauttavat asiaankuuluvan virhekoodin.

5. TEHTÄVIEN TARKASTUKSEN TOTEUTUS

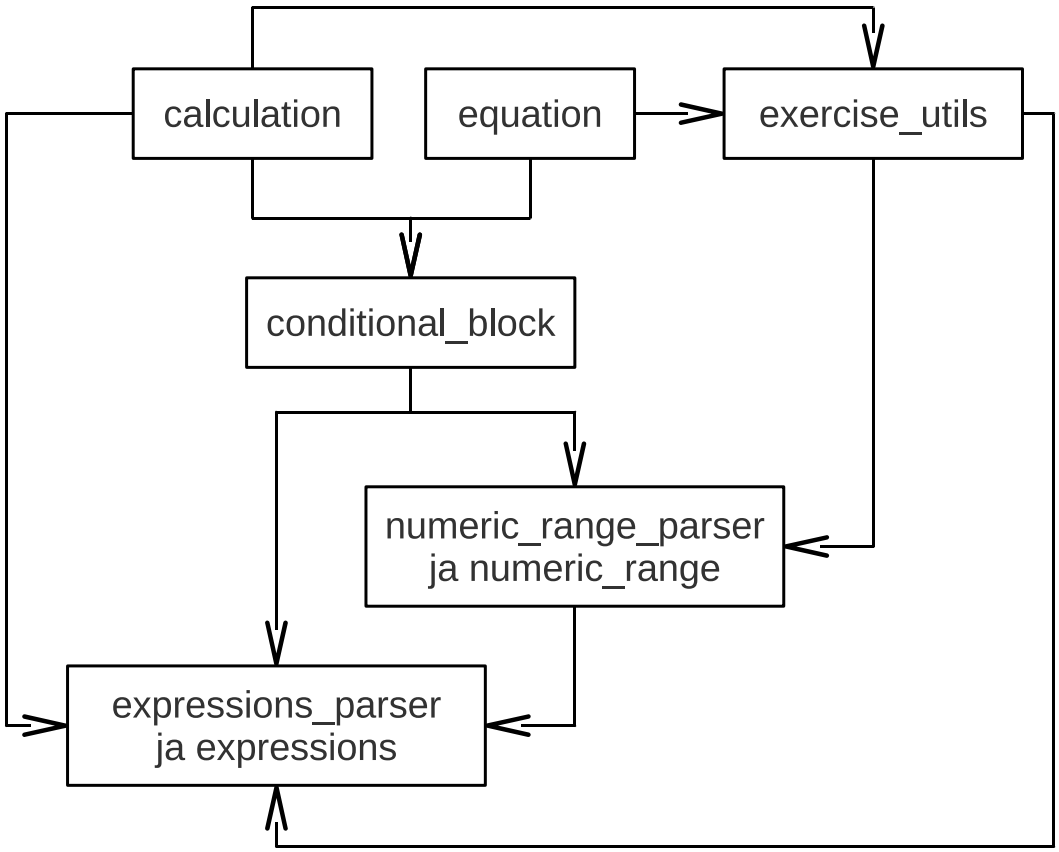
5.1 Paketti *backend*

Tehtävien tarkastuksen toteuttava Python-paketti *backend* on ohjelman suurin ja monimutkaisin osa, joka koostuu useasta moduulista. Tässä aliluvussa käydään läpi moduulien päätehtävät pinnallisesti ja selvitetään niiden väliset riippuvuudet. Seuraavissa aliluvuissa tarkastellaan tärkeimpien moduulien sisäistä toteutusta syvällisemmin. Moduulien väliset suhteet esitetään myös kaaviossa 5.1. Kaaviosta on jätetty selkeyden vuoksi pois muutamia moduuleja, jotka kuitenkin kuvataan tässä luvussa. Kun tekstissä viitataan moduulien tasoihin, tarkoitetaan kaaviossa esitettyä moduulien pystysuuntaista järjestystä. Kuten kaaviosta havaitaan on suunnittelun lähtökohtana ollut, että ylemmän tason moduulit riippuvat vain alemman tason tai saman tason moduuleista.

Moduuli *numeric* sisältää ohjelman oman lukutyyppiluokan *Numeric*, jota käytetään lausekkeiden arvoja laskettaessa. Sen päätehtävä on hoitaa lukualueen rajallisuudesta johtuvat ongelmat. Luokan *Numeric* oliot kuvaavat siis lukuja ja luokan rajapinta tarjoaa kaikki tarvittavat lasku- ja vertailuoperaattorit. Se ei riipu mistään muusta moduulista ja lähes kaikki muut moduulit riippuvat siitä. Selkeyden vuoksi se on jätetty kaaviosta 5.1 pois.

Toinen hyvin perustavaa tasoa oleva moduuli on *tokenizer*, joka sisältää funktion, joka muuttaa merkkijonot tekstialkioiden listaksi. Se siis käy läpi merkkijonon ja tulkitsee sen merkit joko luvuiksi, muuttujiksi, operaattoreiksi tai välimerkeiksi. Sitä käytetään sekä vastauksen että tehtävien määrittelytiedostojen jäsentämisessä. Jos ohjelmaan täytyisi lisätä tuki desimaaliluvuille, se voitaisiin tehdä muokkaamalla pelkästään tätä moduulia, koska merkkijonon numeroiden jäsennys luokan *numeric* olioksi on toteutettu täysin tässä moduulissa. Se tarjoaa myös mahdollisuuden hyvin helposti vaihtaa ja lisätä operaattoreiden ja välimerkkien merkintöjä, koska ne luetaan erillisestä tiedostosta. Moduuli *tokenizer* riippuu vain moduuleista *numeric* ja *utils* ja siitä riippuvat lähes kaikki muut moduulit, joten selkeyden vuoksi myös tämä moduuli on jätetty pois kaaviosta 5.1. Moduuli on pohjimmiltaan hyvin yksinkertainen, joten sitä ei tarkastella sen tarkemmin jatkossa.

Ohjelman keskeisimpiä moduuleita ovat numeerisia ja loogisia lausekkeita käsittelevät moduulit *expressions_parser* ja *expressions*. Nämä moduulit ovat keskenään tiiviisti yhteydessä ja ovat kaksi eri moduulia vain, koska yhdessä ne ovat hanka-



Kuva 5.1: *Backend*-paketin moduulit. Nuoli osoittaa moduulin, josta alkupään moduuli riippuu.

lan iso tiedosto. Moduuli *expressions_parser* sisältää funktiot, jotka lukevat listan tekstialkioita ja jäsentävät sen luokan *ExpressionTree* olioksi, joka on määritelty moduulissa *expressions*. Luokan *ExpressionTree* nimikin jo vihjaa, että jäsennetyt lausekkeet talletetaan puutietorakenteena. Nämä moduulit riippuvat vain moduuleista *numeric* ja *tokenizer*.

Erään tärkeän abstraktion tarjoaa moduuli *numeric_range*, joka sisältää luokan *NumericRange* määrittelyn. Luokan olio kuvaa lukuvälejä ja tarjoaa helpon rajapinnan muun muassa niiden unionin ja leikkauksen muodostamiseen. Kuten lausekkeita käsittelevien moduulien kanssa, myös tälle moduulille on vastinparina moduuli *numeric_range_parser*, jonka funktiot tuottavat *NumericRange*-olioita lukuvälejä kuvaavista tekstialkiolistoista. Nämä moduulit riippuvat moduuleista *numeric*, *tokenizer* ja *expressions_parser*.

Kenties ohjelman keskeisin moduuli on *conditional_block*, joka sisältää luokan *ConditionalBlock* määrittelyn. Yksi luokan *ConditionalBlock* olio kuvaa yhtä ehtolohkoa ja tarjoaa sen tarkasteluun tarvittavat metodit. Luokan rajapinnan tär-

keimmät metodit etsivät virheitä päättelyketjuista ja testaavat kahden luokan olion välistä yhtäsuuruutta tai yhtäpitävyyttä kokeilemalla eri testiarvoja aliluvussa 3.1 kuvatulla tavalla. Hieman hämäävästi tätä luokkaa käytetään myös laskutehtävien toteutuksessa vaikka ehtolohkot ovat käyttäjän käytössä vain yhtälönratkaisutehtävissä. Näin on tehty yksinkertaisesti, koska ehtolohkon osana on päättelyketju, joiden tarkastusperiaate on täysin sama kuin laskutehtävien tarkastuksessa. Molempien tehtävätyyppien tarkastus saadaan toteutettua samalla, kunhan lausekkeiden välisen relaation tarkastus parametrisoidaan sopivalla tavalla. Periaatteessa moduulin nykyinen toteutus mahdollistaisi ehtolohkojen käytön myös laskutehtävissä, mutta ominaisuus on jätetty pois käytettävyyden ja selkeyden vuoksi. Muista poiketen ehtolohkojen jäsennysfunktiot ovat samassa moduulissa. Tämä moduuli käyttää kaikkia tähän asti esiteltyjä moduuleita.

Eri tehtävätyyppien tarkastuksen korkealla tasolla hoitavat funktiot löytyvät tehtävätyyppimoduuleista *calculation* ja *equation*. Moduulit sisältävät myös funktiot, jotka lukevat ja tarkastavat tehtävien määrittelytiedostot. Nämä moduulit pysyvät melko yksinkertaisina, koska ne pystyvät hyödyntämään tehokkaasti *ConditionalBlock*-luokan palveluita. Ne tarvitsevat sen lisäksi suoraan moduuleita *tokenizer*, *exercise_utils* ja *expressions_parser*. Moduuli *exercise_utils* sisältää sekalaisia apufunktioita, joita tehtävätyyppimoduulit tarvitsevat. Se puolestaan tarvitsee moduuleita *utils*, *numeric*, *tokenizer*, *numeric_range_parser*, *numeric_range* ja *expressions_parser*.

Jo kuvattujen lisäksi löytyy moduuli *utils*, joka sisältää lyhyitä apufunktioita, joita vain muutama moduuli käyttää ja se ei itse riipu mistään muusta moduulista. Selkeyden vuoksi se on jätetty kaaviosta 5.1 pois. Python-paketista *backend* käytetään paketin ulkopuolelta funktioita vain moduuleista *calculation*, *equation*, *exercise_utils* ja *utils*.

Tässä luvussa esitetään myös muutamia algoritmeja pseudokoodin tasolla. Pseudokoodi muistuttaa pitkälti Pythonia muun muassa sisennyksen käyttämisessä ehtojen ja silmukoiden rajojen määräämisessä. Muita tuttuja merkintöjä on listan alkioihin viittaus hakasuluilla `[]` ja olion jäsenmuuttujiin tai tietueen kenttiin viittaus pisteoperaattorilla. Listojen pituudet ilmaistaan funktiolla `len()`, joka ottaa parametrinaan listan ja palauttaa sen pituuden. Merkkiä `#` käytetään aloittamaan kommentti, joita käytetään usein korvaamaan koodinpätkä selkeämmällä kuvauksella sen toiminnasta. Pseudokoodissa käytetään myös Pythonista tuttua usean paluuarvon palauttamista yhdestä funktiokutsusta ja useaan muuttujaan sijoittamista samalla rivillä.

5.2 Moduuli *numeric*

Lukualueen rajallisuus aiheuttaa monia ongelmia, joiden hallitseminen on moduulin *numeric* tehtävä. Ensinnäkin ei ole järkevää sallia itseisarvoltaan rajoittamattoman suurien lukujen käyttö vaikka Pythonissa on valmiiksi sille tuki. Jos rajaa ei olisi, liian suurilla luvuilla tarkastus veisi helposti niin kauan aikaa, että se joudutaan keskeyttämään. Lisäksi käyttäjä ei välttämättä ymmärrä sen johtuvan suurista luvuista. Kun asetetaan raja lausekkeissa esiintyville ja lausekkeiden evaluoinnissa syntyville välituloksille, pystytään antamaan selkeä virheilmoitus. Suurien lukujen käyttäminen tuskin tuo pedagogista lisäarvoakaan.

Lukualueen rajallisuus aiheuttaa ongelmia myös välitulosten tarkkuudessa. Välitulosten tarkkuus ei tietenkään ole ongelma, jos välitulokset ovat esimerkiksi kokonaislukuja, mutta on kuitenkin tilanteita, joissa välituloksesta joudutaan tallentamaan rajatun tarkkuuden likiarvo. Tässä ohjelmassa erityisesti murtopotenssit aiheuttavat tämän ongelman. Esimerkiksi luku $2^{1/2} = \sqrt{2}$ on tunnetusti irrationaaliluku, joita ei voi helposti tallentaa muuten kuin likiarvona. Muiden laskuoperaatioiden tulos voidaan tallentaa tarkasti, kunhan operandit ovat tarkkoja. Jakolaskunkaan tulos ei tuota ongelmaa, koska se voidaan helposti tallentaa rationaalilukumuodossa osoittaja ja nimittäjä erikseen kokonaislukuina. Osoittajan ja nimittäjän itseisarvoille joudutaan kuitenkin asettamaan rajat edellisessä kappaleessa mainituista syistä. Rajat voidaan kuitenkin käytännössä asettaa niin suuriksi, että tavanomaisien tehtävien tarkastus onnistuu tarkoilla välituloksilla, kunhan lausekkeissa ei esiinny murtopotensseja. Lisäksi, vaikka välitulokseen tulisi liian suuri osoittaja tai nimittäjä, on siitä helppo antaa täsmällinen virheilmoitus.

Välitulosten rajoitetulla tarkkuudella on paljon suurempi merkitys ohjelman toiminnan kannalta kuin lukujen itseisarvon rajoittamisella, koska vastauksen tarkastus perustuu lopulta lukuarvojen vertailuun molemmissa tehtävätyypeissä. Jos vertailtavat luvut ovat likiarvoja, ei aina voida olla varmoja vertailun oikeasta tuloksesta ja sitä kautta koko vastauksen oikeellisuudesta. Tämän takia on pyrittävä laskemaan mahdollisimman pitkälle tarkoilla arvoilla ja on tiedostettava, että tarkastus ei voi olla täysin varma, kun lausekkeissa esiintyy murtopotensseja.

Kokonaisuudessaan moduulin *numeric* sisältämä lukutyyppiluokka *Numeric* toimii siis seuraavalla tavalla. Aina kun mahdollista *Numeric*-olion lukuarvo tallennetaan tarkasti rationaalilukumuodossa, eli osoittaja ja nimittäjä erikseen kokonaislukuina. Tässä käytetään itse asiassa Pythonin valmista moduulia *fractions*, jossa on toteutettu rationaalilukutyyppi. Valmis toteutus sieventää osoittajan ja nimittäjän automaattisesti käyttäen Euclideen algoritmia [16, s. 857–858]. Lisäksi valmis toteutus pitää nimittäjän aina positiivisena, jolloin luvun merkki on suoraan osoittajan merkki. Jos kahden tarkan *Numeric*-olion välisen laskuoperaation tulosta ei

voida ilmaista tarkasti rationaalilukumuodossa, on tuloksena *Numeric*-olio, joka tallettaa lukuarvonsa liukulukuna, eli likiarvona. Ainoastaan murtopotenssiin korotus on tällainen operaatio. Jos toinen tai molemmat operandit ovat likiarvoja, on myös tulos likiarvo. Lukutyyppiluokka tarjoaa vastauksissa esiintyvien laskuoperaatioiden lisäksi katto- ja lattiapyöristysfunktiot, joita erityisesti moduuli *numeric_ range* tarvitsee.

Luokka *Numeric* sisältää myös kaikki vastauksissa esiintyvät vertailuoperaattorit. Niiden toteutus on suoraviivaista, kun vertaillaan kahta tarkkaa *numeric*-oliota keskenään. Yhtäsuuruus voidaan testata suoraan vertaamalla osoittajia ja nimittäjiä keskenään, koska automaattinen sievennys ja merkin siirto osoittajaan takaavat niiden olevan yksikäsitteisiä. Muissa vertailuissa rationaaliluvut lavennetaan toistensa nimittäjillä ja vertaillaan osoittajia keskenään.

Likiarvoisten *Numeric*-olioiden vertailun johdonmukainen toteutus onkin hankalampaa. Koska vertaillaan keskenään likiarvoja, täytyy yhtäsuuruuden tarkastelussa sallia pieni virhemarginaali. Esimerkiksi kaksi likiarvoa x ja y tulkitaan yhtä suuriksi jos ja vain jos $|x - y| < \epsilon$, missä ϵ on jokin pieni positiivinen luku. Jotta vertailuoperaattorit toimisivat yhdessä mahdollisimman johdonmukaisesti, täytyy virhemarginaali ottaa huomioon myös operaattoreiden `<`, `<=`, `>=` ja `>` toteutuksessa. Esimerkiksi, jos se otetaan huomioon yhtäsuuruusoperaattorissa mutta ei suurempi kuin-operaattorissa, lauseke `x = 0 ja x > 0` on tosi kun $0 < x < \epsilon$, missä ei tietenkään ole järkeä. Vertailuoperaattorien toteutus likiarvoilla pohjautuu seuraaviin määritelmiin:

$$\begin{aligned} \texttt{x = y} &\iff |x - y| \leq \epsilon, \\ \texttt{x <= y} &\iff x \leq y + \epsilon, \text{ ja} \\ \texttt{x >= y} &\iff x \geq y - \epsilon. \end{aligned}$$

Loput operaattorit saadaan toteutettua likiarvoille edellä määriteltyjen operaattoreiden negaatioina:

$$\begin{aligned} \texttt{x /= y} &\iff \neg\ \texttt{x = y}, \\ \texttt{x > y} &\iff \neg\ \texttt{x <= y}, \text{ ja} \\ \texttt{x < y} &\iff \neg\ \texttt{x >= y}. \end{aligned}$$

Näin määriteltynä vertailuoperaattorit noudattavat likiarvoillakin monia tuttuja loogisia sääntöjä kuten $x = 0 \Rightarrow \neg x > 0$, mutta eivät kuitenkaan kaikkia. Esimerkiksi likiarvoisista lausekkeista $l1$, $l2 = l1 + \epsilon$ ja $l3 = l2 + \epsilon$ muodostettu vertailu `l1 = l2 = l3` on tosi, mutta vertailu `l1 = l3` on epätosi.

Moduulissa *numeric* määritellään myös erikseen *Numeric*-oliot *INF* ja *NEG_ INF*,

jotka tulkitaan vertailuoperaattoreissa vastaavasti äärettömyydeksi ja miinus äärettömyydeksi. Ne on lisätty vain helpottamaan moduulin *numeric_range* algoritmien toteutusta, eikä niitä voi käyttää laskuoperaatioissa. Olio *INF* on siis suurempi kuin mikään muu *Numeric*-olio ja vastaavasti *NEG_INF* on pienempi kuin mikään muu *Numeric*-olio. Tulkinnanvarainen kysymys suunnittelussa oli pitäisikö ne tulkita yhtä suuriksi itsensä kanssa. Moduulin *numeric_range* algoritmien toteutuksessa osoittautui kätevämmäksi tulkita ne yhtä suuriksi, joten vertailut toteutettiin sillä periaatteella.

Kaikki lasku- ja vertailuoperaattorit toteutettiin ylikuormittamalla Pythonin omia operaattoreita, jolloin tuloksena oli hyvin helppokäyttöinen rajapinta. Jos ohjelmaan lisätään uusia laskuoperaatioita, kuten esimerkiksi logaritmi, täytyy kaavasta poiketa. Tämän moduulin kohdalla lisäys onnistuu kuitenkin yksinkertaisesti määrittelemällä uusi funktio rajapintaan.

Kuten on tullut ilmi, moduulin *numeric* toimintaa ohjaa muutama vakio. Ensinnäkin rationaalilukujen osoittajien ja nimittäjien itseisarvoilla on ylärajat, jotka samalla asettavat rajan lukujen itseisarvon suuruudelle. Testauksessa osoittautui, että 2^{64} on tarpeeksi suuri raja kirjan [5] tehtävillä. Suurempaa rajaa ei kannata valita, koska se johtaa helposti keskeytyneisiin tarkastuksiin suurilla luvuilla, kuten tämän aliluvun alussa todettiin. Jos moduuli pitäisi toteuttaa uudelleen C++:lla, lukualueen rajan määräisivät C++:n tietotyypit, joista yksi sopii yhteen tämän rajan kanssa. Rajan 2^{64} valinta on siis senkin puolesta sopiva. Mainittakoon samalla, että moduuli käyttää Pythonin liukulukutyyppiä *float*, joka useimmissa Python tulkeissa on tuplatarkkuuden, eli 64 bitin, liukuluku, jolle myös löytyy vastaava tietotyyppi C++-kielestä. Toinen tärkeä vakio määrää likiarvojen vertailun virhemarginaalin. Virhemarginaalin sopiva valinta riippuu täysin tarkastettavista tehtävistä. Jos sen valitsee liian pieneksi, pitävät vertailut tulkitaan virheellisesti pitämättömiksi ja toisaalta, jos sen valitsee liian suureksi, pitämättömät vertailut voidaan tulkita pitäviksi. Kirjan [5] tehtäville arvo 2^{-32} osoittautui sopivaksi.

Moduulissa määritellään myös erityinen *Numeric*-olio *EPS*, jota käytetään testiarvojen valinnassa. Se on siis pieni positiivinen tarkka rationaaliluku, jota käytetään kun tarvitsee muodostaa testiarvo, joka on hieman pienempi tai suurempi kuin jokin luku. Vaikka *EPS* pitäisi ideaalisesti valita mahdollisimman pieneksi, se ei ole käytännössä mahdollista. Jos se valitaan liian pieneksi, testiarvojen ja välitulosten osoittajan ja nimittäjän itseisarvot kasvavat helposti yli rajojensa. Käytännön kokeilut osoittivat, että arvo 2^{-8} toimii kirjan [5] tehtävien kanssa. Mikäli ohjelman täytyy selviytyä suuremmista luvuista tai tarkemmista laskuista, täytyy näitä vakioita säätää tarpeen mukaan.

Aikaisessa kehitysvaiheessa *Numeric*-luokan tilalla käytettiin yksinkertaisesti liukulukutyyppiä *float*. Kun siirryttiin *Numeric*-luokkaan ohjelma hidastui merkittä-

västi. Kuten tämän luvun alussa todettiin, kahden lausekkeen välisen relaation tarkastuksessa lausekkeet saatetaan evaluoida hyvinkin monta kertaa. Lasku- ja vertailuoperaatiot ovat lausekkeiden evaluoinnin perusoperaatioita, joten ne ovat hyvin nopeuskriittisiä. Erityisesti luokan *Numeric* rakentajan optimointi osoittautui tärkeäksi. Operaatiot ovat kuitenkin niin yksinkertaisia, että niissä ei ole paljon optimoimisen varaa. Tämän moduulin toteutus C++:lla on hyvin varteenotettava vaihtoehto mutta Python toteutus osoittautui kuitenkin tarpeeksi nopeaksi. Ohjelman tehokkuuteen kokonaisuutena palataan vielä luvussa 7. Toisaalta, kun *float* korvattiin *Numeric*-luokalla, Pythonin heikon tyypityksen ja operaattoreiden ylikuormituksen ansiosta valmista koodia täytyi muokata vain muutamasta kohdasta, mikä oli osoitus Pythonin hyvistä ominaisuuksista.

5.3 Moduulit *expressions* ja *expressions_parser*

Numeeristen ja loogisten lausekkeiden tietorakenteen toteuttaa moduulin *expressions* sisältämä luokka *ExpressionTree*. Luokan *ExpressionTree* olio voi siis kuvata joko numeerista tai loogista lauseketta. Itse tietorakenne on puu, jossa luvut ja muuttujat ovat lehtisolmuja ja operaattorit sisäsolmuja. Tätä rakennetta kutsutaan jatkossa lausekepuuksi. Lukusolmu sisältää lukuarvonsa *Numeric*-luokan oliona ja muuttujasolmu sisältää muuttujan symbolin merkkijonona. Operaattorisolmut sisältävät viittaukset lapsiinsa, eli alipuihin, jotka kuvaavat operaattorin operandeja. Unaarioperaattoreilla on yksi lapsi, binäärioperaattoreilla kaksi ja vertailut toteuttavilla solmuilla kaksi tai useampia, koska kokonainen vertailuketju esitetään yhtenä solmuna. Vertailuketjusolmun täytyy myös sisältää tieto ketjussa olevien vertailuoperaattorien tyypeistä.

Unaari- ja binäärioperaattoreille on omat kantaluokkansa, jotka sisältävät kaikille samantyyppisille operaattoreille yhteisiä metodeja ja apufunktioita. Jokaista operaattoria varten on vielä oma luokkansa, joka toteuttaa sille operaattorille ominaisen toiminnallisuuden kuten esimerkiksi solmun arvon evaluoinnin. Periyttämisen käyttö tässä tilanteessa pienentää koodin määrää merkittävästi.

Luokka *ExpressionTree* on hyvin yksinkertainen, koska lausekepuiden varsinainen toiminnallisuus on solmuluokissa. Luokka sisältää vain yhden jäsenmuuttujan, joka osoittaa lausekepuun juurisolmuun. Luokan metodit puolestaan vain kutsuvat muutamaa metodia juurisolmulle, joka puolestaan tarvittaessa kutsuu vastaavaa metodia lapsilleen. Esimerkiksi lausekepuun evaluoinnissa jokainen solmu kutsuu lapsiensa evaluointimetodia, jonka jälkeen solmu voi laskea oman arvonsa ja palauttaa sen kutsujalleen ylöspäin puussa.

Evaluoinnin lisäksi luokan *ExpressionTree* tärkeimmät metodit laskevat erityyppisten solmujen lukumääriä ja vertailevat ovatko kaksi lausekepuuta identtiset keskenään. Lisäksi loogista lauseketta kuvaavalle *ExpressionTree*-oliolle voi kutsua paria

```
01 def jäsennäA(l, i):
02   b, i = jäsennäB(l, i)
03   if i < len(l) and l[i] == "%":
04     a, i = jäsennäA(l, i + 1)
05     # Muodosta oikea solmu s muuttujista b ja a.
06     return s, i
07   return b, i
```

Kuva 5.2: Esimerkki jäsennysfunktiosta pseudokoodina.

muuta tärkeää metodia. Yksi metodi tarkastaa, että lauseke on ratkaistussa muodossa annetun muuttujan suhteen ja toinen etsii lausekkeen vertailusolmuista sopivia testiarvoja aliluvussa 3.4 kuvatulla tavalla.

Luokan *ExpressionTree* olioita luodaan kutsumalla moduulin *expressions_parser* funktioita, jotka jäsentävät moduulin *tokenizer* tuottamia tekstialkiolistoja. Uusien operaattoreiden lisäys on hyvin todennäköistä ohjelman jatkokehityksessä, joten sen tekeminen helpoksi on tärkeää. Tästä syystä lausekkeiden jäsentämisen toteutukseen kiinnitettiin erityistä huomiota ja se toteutettiin noudattaen johdonmukaisesti yhtä menetelmää.

Jäsennys perustuu rekursiivisen laskeutumisen (eng. recursive descent) [17, 18] menetelmään. Rekursiivisessa laskeutumisessa jokaista välisymbolia vastaa funktio, joka osaa jäsentää oman välisymbolinsa muiden välisymbolien funktioita apuna käyttäen. Jäsennys aloitetaan kutsumalla aloitussymbolin funktiota ja osoittamalla se tekstialkiolistan alkuun, mistä jäsennys jatkuu kutsumalla muiden välisymbolien funktioita pitäen kirjaa missä kohtaa tekstialkiolistaa ollaan. Funktiot luovat oikean tyyppisen lausekepuun solmun pohjautuen välisymbolin sääntöihin, antavat sille parametrina sen operandit ja palauttavat luomansa solmun kutsujalleen.

Välisymbolien funktioiden kirjoittaminen on hyvin mekaanista, kunhan on olemassa sopivassa muodossa oleva BNF-kuvaus jäsennettävästä kielestä. Tarkastellaan esimerkkiä. Halutaan kirjoittaa välisymbolin `A` jäsentävä funktio, kun

```
A ::= B | B "%"A,
```

jossa `%` on jokin operaattori ja B on toinen välisymboli. Jäsennysfunktion toteutus esitetään kuvassa 5.2, jossa `l` on tekstialkioita vastaava lista, `i` on sen hetkinen paikka listalla ja funktio `jäsennäB` on välisymbolia `B` vastaava funktio.

Tämä on hyvin yksinkertainen ja melko tehokas jäsennin, minkä takia se on hyvä valinta tähän ohjelmaan. Rekursiivisen laskeutumisen jäsentäjä on tehokas erityisesti, kun joka tilanteessa jäsentäjä pystyy päättelemään vain sen hetkistä tekstialkiota tarkastelemalla mitä jäsennetään seuraavaksi. Näin ollen jäsentäjä liikkuu aina

```
laus0 ::= laus1 | laus1 "tai" laus0
laus1 ::= laus2 | laus2 "ja" laus1
laus2 ::= laus3 | laus3 (("="|"/="|"<"|"<="|">="|">") laus3)+

laus3 ::= laus4 | laus4 "+" laus3 | laus4 "-" laus3
laus4 ::= laus5 | laus5 "*" laus4 | laus5 laus4
                | laus5 "/" laus4
# Kolmas vaihtoehto yllä olevassa säännössä pätee vain kun laus4
# alkaa luvulla, muuttujalla tai sululla ja ei päde, jos laus5
# loppuu ja laus4 alkaa luvulla.
laus5 ::= laus6 | "+" laus5 | "-" laus5
laus6 ::= laus7 | laus7 "^" laus5
laus7 ::= laus8 | "(" laus0 ")" | "|" laus3 "|"
laus8 ::= "luku" | "muuttuja"
```

Kuva 5.3: Lausekkeiden jäsentämisen toteutuksessa käytetty kielioppi.

eteenpäin tekstialkioiden listassa ja sen suoritusaika on lineaarinen listan pituuteen nähden.

Toisaalta, mitä tahansa BNF:llä kuvattua kielioppia ei voi jäsentää tällä menetelmällä. Luvussa 3 esitettyjä BNF-kielioppeja täytyy hieman muokata, jotta rekursiivisen laskeutumisen menetelmä toimii. Numeeristen ja loogisten lausekkeiden jäsennys on myös yhdistetty, koska numeeriset lausekkeet ovat osa loogisten lausekkeiden syntaksia. Lausekkeiden jäsennys perustuu kuvassa 5.3 esitettyyn kielioppiin, jonka pohjalta pystytään toteuttamaan yksinkertainen jäsennin, joka päättelee seuraavaksi jäsennettävän välisymbolin vain sen hetkisen tekstialkion perusteella.

Kieliopissa on muutamia tahallisia epäkohtia, joiden huomioon ottaminen osoittautui helpommaksi kieliopin ulkopuolella. Ensinnäkin kaikki rekursiot on asetettu oikealle rekursioiksi. Rekursiivisen laskeutumisen menetelmä ei toimi säännöille, jossa ensimmäinen merkki on säännön välisymboli itse. Esimerkiksi sääntöä `A ::= A "%" B` ei voi jäsentää, koska jäsentäjä joutuu loputtomaan rekursioon, jossa yritetään jäsentää välisymbolia `A`. Tämän muutoksen takia vähennys- ja jako-operaattorin sitovuudet ovat väärin päin. Sitovuudet korjataan jäsennyksen jälkeen omassa jälkikäsittelyvaiheessaan. Vähennyslaskun sitovuuden korjaava algoritmi esitellään kuvassa 5.4. Jakolaskun sitovuuden korjaava algoritmi on täysin vastaava, joten sitä ei erikseen esitellä. Merkinnät `s.vasen` ja `s.oikea` viittaavat binäärioperaattorisolmun `s` vasempaan ja oikeaan lapseen.

Algoritmi aloitetaan siis kutsumalla funktiota `korjaaVähSit()` parametrina `s` lausekepuun juuri. Siitä algoritmi jatkaa vasemmanpuoleisiin lapsiin ja korjaa niiden sitovuudet ensin. Funktio palauttaa korjatun alipuun juuren, joka saattaa olla eri solmu kuin mitä annettiin parametrina, joten funktion paluuarvo pitää aina si-

```
01 def korjaaVähSit(s):
02   # Jos s ei ole binäärioperaattorin solmu:
03     # Kaikille solmun s lapsille s.l:
04       s.l = korjaaVähSit(s.l)
05   else:
06     s.vasen = korjaaVähSit(s.vasen)
07     # Niin kauan kuin s on vähennyslaskusolmu ja
08     # ja s.oikea on vähennys- tai yhteenlaskusolmu:
09       tmp = s.oikea
10       s.oikea = tmp.vasen
11       tmp.vasen = s
12       s = tmp
13       s.vasen.oikea = korjaaVähSit(s.vasen.oikea)
14     s.oikea = korjaaVähSit(s.oikea)
15   return s
```

Kuva 5.4: Vähennyslaskun sitovuuden korjaus pseudokoodina.

joittaa uudeksi lapseksi. Virheellisen vähennyslaskusolmun tunnistaa siitä, että sen oikea lapsi on vähennys- tai yhteenlaskusolmu. Lausekepuussa on omat solmunsa sulkulausekkeille, joten niiden määräämää laskujärjestystä ei rikota. Jos solmu `s` on virheellinen vähennyslaskusolmu, suoritetaan binäärihakupuista tuttu vasemmalle rotaatio [16, s. 277–278] riveillä 9–12. Tässä yhteydessä täytyy muistaa, että rotaatiossa yksi solmu siirtyy uuden juuren vasempaan alipuuhun, eikä sitä ole vielä korjattu. Tämä hoidetaan algoritmin rivillä 13. Täytyy muistaa myös, että solmun `s` paikalle siirretty uusi solmu voi olla virheellinen, minkä takia riveillä 7 ja 8 on silmukka eikä ehtolauseke. Kun solmu `s` ei ole enää virheellinen jatketaan rekursiota oikeaan lapseen. Lopuksi palautetaan vielä korjatun alipuun mahdollisesti uusi juuri `s`. Varsinaisessa toteutuksessa vähennys- ja jakolaskun sitovuuksien korjaus on yhdistetty samaan funktioon, jolloin lausekepuu täytyy käydä vain kerran läpi.

Aivan toisenlainen lähestymistapa sitovuusongelmaan olisi luopua rekursion käytöstä ja toteuttaa välisymbolien `laus3` ja `laus4` jäsennys silmukkarakenteella. Esimerkiksi `laus3` kohdalla silmukka siis jäsentäisi samassa funktiokutsussa kaikki saman tason yhteen- ja vähennyslaskut ja sitten muodostaisi sitovuudet oikein huomioon ottavan alipuun. Tällä tavalla päästäisiin eroon jälkikäsittelyvaiheesta ja se olisi hyvä lähestymistapa erityisesti, jos halutaan käyttää usean termin summaa ja vähennystä esittäviä solmuja.

Toinen selvä epäkohta on, että kielioppi 5.3 sallii numeeristen lausekkeiden käytön loogisten operaattoreiden operandeina ja toisinkin päin kun loogiset lausekkeet ympäröidään suluilla. Lausekkeiden tyypin tarkastus suoritetaan jäsentämisen aikana aina ennen kuin operandit annetaan operaattorisolmun parametreiksi. Kieli-

opin laatiminen sellaiseksi, että se takaa operaattoreille oikeantyyppiset parametrit osoittautui hyvin haastavaksi. Erityisesti sulut ovat ongelmallisia, koska niitä pitää voida käyttää sekä numeeristen että loogisten lausekkeiden kanssa. Jos kieliopissa määriteltäisiin loogisille ja numeerisille lausekkeille erikseen sulut samalla merkinnällä, joudutaan ongelmiin kun jäsentäjä saa loogista lauseketta jäsentäessään vastaan aloittavan sulun. Jäsentäjä ei voi siihen asti lukemistaan tekstialkioista päätellä kuuluuko se loogiseen vai numeeriseen lausekkeeseen. Tämän ongelman voisi kiertää käyttämällä loogisille lausekkeille vain aaltosulkeita, mutta se ei vastaisi paperilla laskemisen käytäntöjä. Lausekkeiden tyypin tarkastaminen erikseen ei lopulta lisää koodin määrää huomattavasti ja tuloksenakin on yksinkertaisempi kielioppi.

Syntaksivirheiden käsittelyssä käytettiin hyväksi poikkeuksia, koska ne sopivat erinomaisen hyvin tällaisen rekursiivisen algoritmin virheenkäsittelyyn. Kuten luvussa 4 mainittiin, syntaksivirheistä pitää lopulta ilmoittaa palauttamalla virhekoodi, mutta ei ole järkevää käyttää paluuarvoja virheenkäsittelyyn jäsennysfunktioiden tasolla. Jos virheitä käsiteltäisiin paluuarvoina, koodia kertyisi paljon enemmän ja siitä tulisi hankalampaa lukea, koska paluuarvo pitäisi tarkastaa hyvin monessa kohtaa koodia. Poikkeuksilla puolestaan syntaksivirheiden käsittely voidaan keskittää yhteen funktioon, joka nappaa poikkeukset ja palauttaa sitä vastaavan virhekoodin.

5.4 Moduulit *numeric_range* ja *numeric_range_parser*

Kuvassa 3.5 esitetyllä syntaksilla määriteltävissä olevia lukuvälejä vastaa toteutuksessa moduulin *numeric_range* sisältämä luokka *NumericRange*. Luokan nimi on hieman harhaanjohtava, sillä luokka vastaa useata lukusuoran väliä vaikka nimi viittaa vain yhteen väliin. Koodissa haluttiin noudattaa ohjelmointikäytäntöä, jossa luokkien nimet ovat yksikössä. Luokka *NumericRange* tarjoaa muun muassa unionin, leikkauksen ja vertailun kahden *NumericRange*-olion kesken. Vertailussa kaksi *NumericRange*-oliota ovat keskenään samat kun ne kuvaavat täsmälleen samoja lukuvälejä lukusuoralla.

Luokalla on jäsenmuuttujina muuttujasymboli, jota lukuväli koskee, ja lista tietueita, jotka kuvaavat yhtenäisiä lukuvälejä lukusuoralla. Luku x kuuluu *NumericRange*-olion lukuväleihin jos ja vain jos luku x kuuluu yhteenkin listan lukuväleistä. Lukuvälitietue sisältää välin ala- ja ylärajan *Numeric*-olioina ja rajoja vastaavat totuusarvot, jotka kertovat kuuluuko raja lukuväliin. Alaraja ei saa koskaan olla suurempi kuin yläraja mutta ne voivat olla yhtä suuret, jolloin lukuväli kuvaa yhtä pistettä. Yläraja voi olla myös *INF* ja alaraja voi olla *NEG_INF*, jolloin tulkinta on tietenkin, että väli jatkuu äärettömyyteen vastaavista päistä. Molemmat rajat voivat olla myös samaan aikaan äärettömyydessä, jolloin *NumericRange*-olio vastaa koko lukusuoraa. Tyhjä lukuväli puolestaan ilmaistaan tyhjänä listana. Rajojen täytyy tietenkin kuulua väliin pisteen tapauksessa ja äärettömissä tapauksissa rajat

eivät saa kuulua väliin.

Lukuvälien lista pidetään tietyssä normalisoidussa muodossa, jossa päällekkäisiä tai toisiaan sivuavia lukuvälejä ei ole ja lukuvälit on järjestetty lukusuoran negatiivisesta päästä alkaen. Ensinnäkin tämä mahdollistaa kahden *NumericRange* olion helpon vertailun keskenään, sillä normalisoitu lista on yksikäsitteinen. Toiseksi operaatiot kuten leikkaus ja lukuväleihin kuuluvuuden testaus voidaan toteuttaa tehokkaammin.

Kahden *NumericRange*-olion unioni on toteutettu yhdistämällä niiden lukuvälien listat ja sitten normalisoimalla sen tulos. Ensimmäisenä normalisointialgoritmi järjestää listan alarajojen mukaan. Jos usealla välillä on sama alaraja, laitetaan ensin ne, joilla alaraja kuuluu väliin. Tämän jälkeen algoritmi käy listan välit läpi järjestyksessä tarkastellen kerralla kahta peräkkäistä väliä. Jos välit eivät mene päällekkäin tai sivua toisiaan, tarkasteltavista ensimmäinen siirretään lopputulokseen ja siirrytään listassa eteenpäin. Jos välit menevät päällekkäin tai sivuavat toisiaan, ne yhdistetään ja seuraavalla kierroksella yhdistetty väli on ensimmäisen välin paikalla ja toisen välin paikalle otetaan seuraava väli listasta. Tätä jatketaan kunnes lista on käyty läpi, jolloin tuloksena on normalisoidussa muodossa oleva lista.

Tämä ei ole tehokkain mahdollinen tapa toteuttaa unionia. Listan järjestäminen on algoritmin heikoin lenkki, koska se vie parhaillakin järjestämisalgoritmeilla $O(n \log n)$ aikaa, kun muu algoritmi on kertaluokassa $O(n)$, missä n on listan pituus. Listan järjestämisen voisi välttää toteuttamalla algoritmi, joka käy molempia normalisoituja listoja läpi samanaikaisesti. Käytännössä lukuvälilistat sisältävät kuitenkin vain muutaman välin, joten tyydyttiin ensimmäisenä kuvattuun ratkaisuun.

Leikkaus sen sijaan toteutettiin tehokkaammalla algoritmilla, joka käy läpi molempien *NumericRange*-olioiden lukuvälilistat samanaikaisesti. Leikkaus ei ole tehokkuudeltaan sen kriittisempi kuin unioni, mutta sen toteutukseen ei ollut muuta selvästi helpompaa tapaa. Algoritmi esitetään kuvassa 5.5. Välien kentät `ala` ja `ylä` vastaavat välin ala- ja ylärajaa ja kentät `alaMukana` ja `yläMukana` totuusarvoa, joka on tosi kun vastaava raja kuuluu väliin.

Algoritmi käy läpi kumpaakin lukuvälilistaa silmukkamuuttujilla `i1` ja `i2`, jotka osoittavat lukuvälilistojen väleihin `v1` ja `v2` vastaavasti. Rivillä 8 tarkastetaan leikkaavatko välit useassa pisteessä. Jos välit leikkaavat, leikkaus muodostetaan ja lisätään tulokseen rivillä 9. Leikkauksen muodostus tässä vaatii melko pitkän ehtorakennelman, joten sitä ei selkeyden vuoksi esitetä pseudokoodissa. Riveillä 10–13 puolestaan hoidetaan tapaukset, joissa välit leikkaavat vain yhdessä pisteessä, eli toisessa välin `v1` rajoista. Riveillä 14–18 siirrytään eteenpäin yhdessä listassa tai molemmissa samaan aikaan. Väli, jonka yläraja on pienempi, voidaan unohtaa, koska se ei voi enää leikata minkään muun toisen listan loppuosasta löytyvän välin kanssa. Kaikkien loppuosasta löytyvien välien alaraja on suurempi kuin sen hetkisen

```
01 def leikkaaVälit(välit1, välit2):
02   välit = [] # Tyhjän listan alustus.
03   i1 = 0
04   i2 = 0
05   while i1 < len(välit1) and i2 < len(välit2):
06     v1 = välit1[i1]
07     v2 = välit2[i2]
08     if v1.ylä > v2.ala and v1.ala < v2.ylä:
09       # Lisätään v1:n ja v2:n leikkaus listaan välit.
10     elif v1.ylä == v2.ala and v1.yläMukana and v2.alaMukana:
11       # Lisätään v1:n yläraja listaan välit.
12     elif v1.ala == v2.ylä and v1.alaMukana and v2.yläMukana:
13       # Lisätään v1:n alaraja listaan välit.
14     if v1.ylä == v2.ylä:
15       i1 += 1
16       i2 += 1
17     elif v1.ylä < v2.ylä: i1 += 1
18     else: i2 += 1
19   return välit
```

Kuva 5.5: Lukuvälilistojen leikkausalgoritmin pseudokoodi.

välin yläraja, joka on puolestaan suurempi kuin hylättävän välin yläraja. Erikoistapauksessa, jossa ylärajat ovat yhtä suuret, molemmat välit voidaan unohtaa, koska kumpikaan ei voi leikata toisen listan loppuosan välien kanssa. Riveillä 14–18 vähintään toinen silmukkamuuttuja kasvaa yhdellä, joten algoritmin suoritusaika on $O(n1 + n2)$, jossa $n1$ ja $n2$ ovat listojen pituudet.

Muihin luokan *NumericRange* metodeihin kuuluu muun muassa tarkastus kuuluuko annettu luku lukuväleihin. Se on toteutettu yksinkertaisesti käymällä läpi lukuvälien lista järjestyksessä. Toteutuksessa voisi hyödyntää puolitushaun tapaista menetelmää, sillä Python-listat ovat itse asiassa sisäiseltä toteutukseltaan taulukoita ja tarjoavat siten vakioaikaisen indeksoinnin. Tämä metodi ei kuitenkaan ole kovin tehokkuuskriittinen, joten tyydyttiin tehottomampaan ratkaisuun. Toinen melko tärkeä metodi valitsee lukuväleistä testiarvoja aliluvussa 3.3 kuvatulla tavalla. Metodin toteutus osoittautui melko monimutkaiseksi ja se olikin lopulta koodin määrässä suurin metodi tässä luokassa. Moduuli *numeric_range* sisältää myös erityiset funktiot, joilla voi luoda koko lukusuoran kattavia ja tyhjää lukuväliä kuvaavia *NumericRange*-olioita.

Lukuvälien syntaksin kuvassa 3.5 jäsentävät funktiot löytyvät moduulista *numeric_range_parser*. Myös tämä jäsentäjä on toteutettu periaatteella, jossa jokaista BNF-kuvauksen välisymbolia vastaa yksi jäsennysfunktio. Tätä jäsentäjää ei

ole kuitenkaan toteutettu rekursiivisen laskeutumisen menetelmällä, kuten *expressions_parser*-moduulin jäsentäjää. Muutama jäsennysfunktio käy koko tekstialkioiden listan läpi, joten se ei ole yhtä tehokas kuin *expressions_parser*-moduulin jäsentäjä. Paremman jäsentäjän laatiminen katsottiin rajallisen aikataulun puitteissa toissijaiseksi muiden ohjelman osien parantamiseen nähden. Heikko tehokkuus on tuskin ongelma, koska lukuvälilausekkeet ovat usein hyvin lyhyitä. Moduulia *expressions_parser* vastaavalla tavalla myös tämä moduuli käyttää poikkeuksia syntaksivirheiden käsittelemiseen jäsennysfunktioiden tasolla.

5.5 Moduuli *conditional_block*

Moduulin *conditional_block* sisältämä luokka *ConditionalBlock* toteuttaa ehtolohkorakenteen. Kuten aliluvussa 3.4 mainittiin, ehtolohkot muodostuvat kahdesta osasta. Päättelyketju talletetaan listana *ExpressionTree*-olioita ja niiden välillä relaatiotekstialkioita eli käytännössä joko yhtäsuuruuksia tai yhtäpitävyyksiä. Haarauma talletetaan puolestaan listana tietueita, jotka kukin kuvaavat yhtä haaraa. Tietue sisältää muun muassa haaran arvoalueen muuttujalle *NumericRange*-oliona ja haaran sisällön *ConditionalBlock*-oliona. Jos ehtolohko sisältää sekä päättelyketjun että haarauman, täytyy päättelyketjulistan loppua relaatiotekstialkioon.

Luokka *ConditionalBlock* kuvaa molempien tehtävätyyppien vastausta korkeimmalla syntaksin tasolla. Laskutehtävissä vastaus on yksi ehtolohko, jolla on vain päättelyketju ja ei haaraumaa. Koska luokka *ConditionalBlock* kuvaa syntaksin korkeinta tasoa, se tarjoaa monia tärkeitä metodeja, jotka ovat suurimmaksi osaksi läpikutsuja lauseketasolle. Esimerkiksi yksi metodi tarkastaa ovatko vastauksen viimeiset lausekkeet ratkaistussa muodossa. Viimeisillä lausekkeilla viitataan koko ehtolohkohierarkian pohjimmaisten ehtolohkojen päättelyketjujen viimeisiin lausekkeisiin. Toinen tärkeä metodi etsii kaikista päättelyketjun lausekkeista testiarvoja *ExpressionTree*-olion metodia hyväksi käyttäen, kutsuu itseään rekursiivisesti kaikille haaroille ja lopuksi yhdistää ja palauttaa tulokset.

Yksi tärkeä metodi, joka toteutetaan varsinaisesti tässä moduulissa, tarkastaa, että ehtolohkojen haaraumat kattavat ehtolohkon muuttujan koko arvoalueen. Tämän suorittava algoritmi kuvailtiin jo aliluvussa 3.4. Se pystytään toteuttamaan helposti yhdellä rekursiivisella funktiolla, kun käytössä on luokassa *NumericRange* toteutetut unioni-, leikkaus- ja vertailuoperaatiot.

Luokan tärkeimmät metodit suorittavat ehtolohkojen sisäisten relaatioiden tarkastuksen ja kahden ehtolohkon välisen relaation tarkastuksen testiarvoja kokeilemalla. Tässä ohjelmassa relaatio on joko yhtäsuuruus tai yhtäpitävyys, mutta nämä metodit on suunniteltu niin, että relaatio voi olla mikä vain. Molemmille metodeille annetaan parametrina funktio, jolle annetaan parametrina kahdelle lausekkeelle joillakin testiarvoilla evaluoidut arvot ja joka palauttaa päteekö tietty relaatio näillä

arvoilla. Ehtolohkon sisäiset relaatiot tarkastavalle metodille annetaan itse asiassa parametrina tietorakenne, josta se etsii relaation tekstialkion perusteella oikean tarkastusfunktion. Pelkästään näitä funktioita vaihtamalla saa siis testattua eri relaatioita.

Molemmat metodit hyödyntävät useita apufunktioita, jotka pilkkovat tarkastuksen pienemmiksi osakokonaisuuksiksi ottaen huomioon ehtolohkojen määräämät muutokset muuttujien arvoalueissa. Esimerkiksi molemmat käyttävät apufunktiota, joka tekee tarkastuksen yhden lausekkeen ja yhden haarauman kaikkien haarojen välillä.

Varsinaisen tarkastuksen suorittaa lopulta funktio, joka tarkastaa relaation pitävyyden kahden lausekkeen välillä. Ensimmäisenä tämä funktio valitsee jokaiselle muuttujalle halutun määrän testiarvoja. Etusijalla ovat tietenkin parametrina saadut ennalta määrätyt testiarvot. Jos ennalta määrättyjä testiarvoja ei ole tarpeeksi, loput muuttujan testiarvot valitaan käyttäen *NumericRange*-luokassa toteutettua funktiota, jonka toiminta kuvattiin aliluvussa 3.3. Tämän jälkeen funktio käy silmukassa läpi kaikki testiarvojen kombinaatiot, evaluoi niillä lausekkeiden arvot ja testaa parametrina saadulla funktiolla päteekö haluttu relaatio. Jos löytyy yksikin kombinaatio, jolla relaatio ei päde, keskeytetään koko tarkastus ja palautetaan oikea virhekoodi. Testiarvojen kombinaatioita on n^s, kun muuttujia on s ja joka muuttujalla on n testiarvoa. Kombinaatiot läpi käyvä silmukka ja sen kutsumat funktiot ovat siis erittäin tehokkuuskriittisiä varsinkin suurilla määrillä muuttujia.

Ehtolohkojen jäsennys on toteutettu samassa moduulissa itse *ConditionalBlock*-luokan toteutuksen kanssa. Ehtolohkojen syntaksi on sen verran yksinkertainen runsaiden välimerkkien ansiosta, että se toteutettiin vain kahdella funktiolla sen sijaan, että jokaiselle välisymbolille olisi oma funktionsa. Jäsentäjä ei ole kuitenkaan tehokkain mahdollinen, sillä se kerää lausekkeet ja lukuvälien määrittelyt omiin listoihinsa ja kutsuu näille vastaavia jäsentäjiä erikseen. Jälleen kerran voidaan kuitenkin todeta, että jäsennyksen tehokkuus ei ole ohjelman tärkeimpiä ominaisuuksia, joten kehityksessä keskityttiin muihin ohjelman osiin.

5.6 Moduulit *calculation* ja *equation*

Kumpikin moduuleista *calculation* ja *equation* sisältää kolme funktiota, joita kutsutaan käyttöliittymän puolelta. Nämä funktiot lukevat tehtävien määrittelytiedostoja, tarkastavat määritellyn tehtävän vastauksen ja tarkastavat vapaan tehtävän vastauksen. Kaikki kolme funktiota käyttävät aikakatkaisua, jonka toteuttavat apufunktiot ovat moduulissa *exercise_utils*. Määrittelytiedostojen lukufunktiot tekevät myös tarkastuksia sen sisällölle, mikä saattaa kestää hyvinkin kauan, minkä takia myös nämä funktiot ovat aikakatkaisun piirissä. Käytännössä määrittelytiedoston sisältöä ei tarvitsisi joka kerta tarkastaa, mutta älykkäämmän järjestelmän tekeminen

tätä prototyyppiä varten katsottiin toissijaiseksi.

Vastausten tarkastusfunktioiden toiminta kuvattiin hyvin perusteellisesti jo aliluvuissa 3.3 ja 3.4. Niiden toteutus on hyvin yksinkertainen, joten toteutusta ei sen tarkemmin kuvata tässä luvussa. Alusta asti suunnitteluperiaatteena oli, että näissä moduuleissa ei olisi monimutkaisten algoritmien toteutuksia. Sen sijaan ne toteutetaan muissa moduuleissa siten, että ne olisivat mahdollisimman monikäyttöisiä ja hyödyllisiä mahdollisesti jälkeenpäin lisättävien tehtävätyyppien toteutuksessa.

Tehtävien määrittelytiedostojen tarkastuksessa kaikki lausekkeet ja ehtolohkot jäsennetään syntaksivirheiden varalta. Muuttujien arvoalueiden ja tehtävän laatijan määräämien testiarvojen jäsennys on toteutettu omina funktioina moduulissa *exercise_utils*. Lisäksi tehdään tarkastuksia eri kenttien arvojen kesken. Esimerkiksi laskutehtävissä tarkastetaan, että mahdollinen aloituslauseke on yhtä suuri lopullisen lausekkeen kanssa. Näissä tarkastuksissa ilmi tulleista virheistä ilmoitetaan tehtävien laatijalle tarkoitetuilla virhekoodeilla.

6. WEB-KÄYTTÖLIITTYMÄ

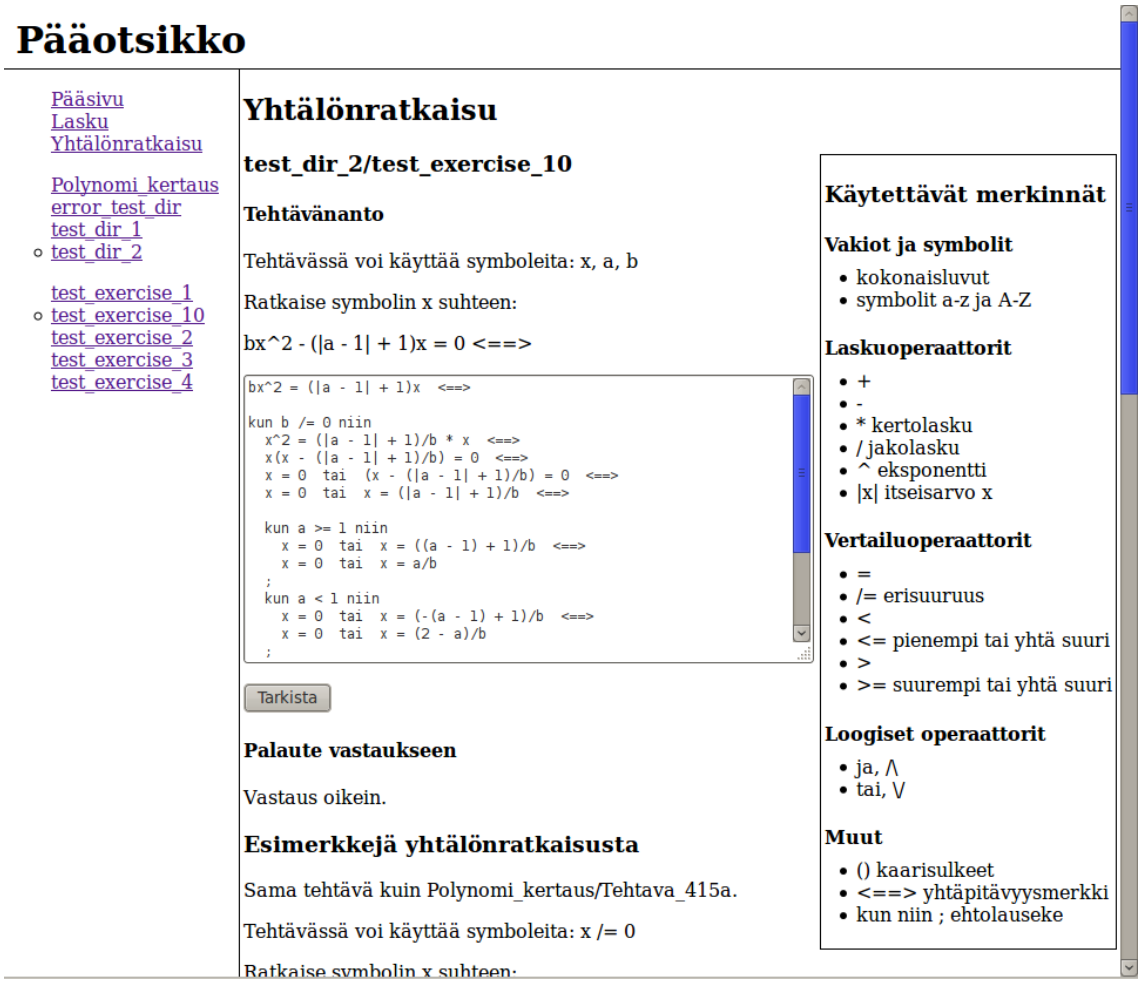
Web-käyttöliittymä koostuu kahdesta perusnäkymästä, joissa molemmissa on vasemmalla laidalla navigaatiovalikko. Toinen näkymä on pääsivu, joka toimii tässä prototyypissä vain lähtöpisteenä. Toinen näkymä on tehtäväsivu, jolla sekä määritellyt että vapaat tehtävät esitetään käyttäjälle ja jolla vastaukset annetaan. Sen ulkomuoto riippuu hiukan tehtävätyypistä mutta pääpiirteissään se on sama kaikille tehtävätyypeille. Esimerkki tehtäväsivusta nähdään kuvassa 6.1, jossa on avoinna yhtälönratkaisutehtävä, jonka vastaus on tarkastettu ja todettu oikeaksi.

Tehtäväsivun oikeassa laidassa esitetään tehtävätyypissä käytössä olevat operaattorit ja merkinnät. Tehtävän tyyppi, tehtävän nimi ja tehtävänanto esitetään sivun keskellä ylhäällä. Heti niiden alapuolella on kenttä, johon vastaus syötetään, ja nappi, joka lähettää vastauksen tarkastettavaksi. Tekstimuotoinen palaute tarkastuksesta esitetään näiden alapuolella. Jos löytyy virhe, joka voidaan paikallistaa tiettyyn kohtaan vastauksessa, virheen paikka osoitetaan itse vastauksesta lisäämällä kysymysmerkki virhekohtaan. Virhekohta ilmoitetaan myös palautetekstissä rivi- ja sarakenumerona. Näiden alapuolella esitetään esimerkkejä samantyyppisistä tehtävistä esimerkkivastauksineen. Todellisessa käytössä ohjelma vaatisi tietenkin paljon kattavammat ohjeet tehtävätyypeistä mutta tässä prototyypissä esimerkit katsottiin riittäviksi.

Kuvassa 6.1 nähdään myös navigaatiovalikon rakenne. Ylimmäisestä kolmen linkin ryhmästä pääsee pääsivulle ja vapaiden tehtävätyyppien tehtäväsivuille. Seuraavaksi alempi linkkien ryhmä sisältää tehtäväryhmien linkit. Ohjelmaan määritellyt tehtävät laitetaan omiin hakemistoihinsa, jotka vastaavat navigaatiovalikossa esitettyjä tehtäväryhmiä. Tämä toiminnallisuus on aika välttämätön käytettävyyden kannalta, koska muuten tietyn tehtävän löytäminen suuresta määrästä tehtäviä olisi hyvin hankalaa. Alimmainen linkkien ryhmä vastaa valitun tehtäväryhmän tehtäviä.

Web-käyttöliittymän toteutus onnistui melko vaivattomasti Pythonin valmiilla CGI-moduulilla. Se koostuu neljästä moduulista ja niistäkin yksi sisältää vain apufunktioita muille moduuleille. Lopuista moduuleista yksi vastaa pääsivun, toinen tehtäväsivun ja kolmas navigaatiovalikon tuottamisesta. Kulloinkin avoinna oleva tehtäväryhmä ja tehtävä välitetään web-palvelimelle GET menetelmällä ja tehtävien vastaukset puolestaan välitetään POST menetelmällä.

Käyttöliittymä toteutettiin siten, että kaikki käyttöliittymässä näkyvät tekstit ja



Kuva 6.1: Tehtäväsivu yhtälönratkaisutehtävällä.

kiinteät HTML-elementit ovat omissa tiedostoissaan, jotka puolestaan ovat omassa hakemistossaan. Tämän johdosta ohjelman voisi kääntää toiselle kielelle tekemällä siinä hakemistossa olevista tiedostoista toisenkieliset versiot. Käyttöliittymässä käytettiin myös CSS-tekniikkaa [19], jonka ansiosta sivujen taittoa voi muokata hyvin pitkälle koskematta koodiin.

Tässä projektissa keskityttiin enemmän varsinaiseen tarkastamiseen kuin käyttöliittymään, joten sen toiminnassa voisi olla paljon parantamisen varaa. Täysin tekstipohjainen käyttöliittymä ei välttämättä ole paras mahdollinen, koska syntaksin oppiminen on oma haasteensa. Ohjelmointia tuntevalle tai muita matemaattisia ohjelmia käyttäneelle syntaksin oppimisen pitäisi olla helppoa, mutta lukiotason oppilailta tällaista ei vielä voi olettaa. Rakenteisten päättelyketjujen käytöstä opetuksessa on tehty tutkimuksia, joissa on tutkittu myös matemaattisen tekstin kirjoittamista tietokoneella [10]. Tutkimuksissa on käynyt ilmi, että oppilaat käyttävät mieluummin hiiripohjaista käyttöliittymää lausekkeiden syöttämiseen. Toinen parannus käyttöliittymään olisi lisätä syötetyn lausekkeen esittäminen erillisessä ku-

vassa oikeilla matemaattisilla merkinnöillä, jolloin käyttäjä huomaisi helpommin virheet syöttämässään lausekkeessa. Tämän toteutus olisi todennäköisesti hieman helpompaa ja toisaalta hyvä hiiripohjainen käyttöliittymä todennäköisesti vaatisi myös tällaisen kuvan.

7. OHJELMAN TEHOKKUUS

Vaikka ohjelman tehokkuus ei ollutkaan tämän projektin tärkein päämäärä, kannattaa sitä hieman tarkastella kokonaisuutena mahdollisen jatkokehityksen kannalta. Ohjelman suorituskyvyn vaatimuksiin vaikuttaa pääosin kaksi asiaa. Ensinnäkin täytyy ottaa huomioon kuinka monta samanaikaista käyttäjää ohjelmalla voi olla. Tämän projektin tavoitteena oli tehdä prototyyppi, jolla voisi suorittaa noin kymmenen käyttäjän koekäytön, mutta esimerkiksi luokkatilanteessa voisi samanaikaisia käyttäjiä olla useita kymmeniä. Toiseksi tarkastettavien tehtävien ja vastausten monimutkaisuus vaikuttaa merkittävästi yhden tarkastuksen kestoon.

Suuri määrä samanaikaisia käyttäjiä vaatii itse tarkastuksen tehokkuuden lisäksi ohjelman web-puolen toteutukselta tehokkuutta. CGI-tekniikan tunnettu heikko puoli on, että jokaisella HTML-palvelupyynnöllä web-palvelin käynnistää CGI-ohjelman uudestaan. Tässä ohjelmassa se tarkoittaa Python-tulkin käynnistämistä ja lisäksi itse ohjelmassa tehtäviä alustuksia. Erityisesti Python joutuu lataamaan kaikki ohjelman moduulit ennen kuin aloittaa varsinaisen suorituksen, mikä vie merkittävästi aikaa. Ohjelman tehokkuutta samanaikaisten käyttäjien suhteen ei testattu käytännössä, koska testipalvelimella yksi tavanomainen tarkastus kesti alle sekunnin. Vain kymmenellä käyttäjällä on hyvin epätodennäköistä, että moni käyttäjä sattuu tarkastuttamaan vastauksensa juuri samaan aikaan kun tarkastus ei kestä sekuntiakaan.

Jos tehokkuus muodostuisi ongelmaksi suurilla käyttäjämäärillä, kannattaisi luultavasti ensimmäisenä korvata CGI-tekniikka tehokkaammalla vaihtoehdolla. Esimerkiksi FastCGI-nimellä tunnettu tekniikka toimii tehokkaammin käynnistämällä web-sovelluksen vain kerran [20]. Web-sovellus on siis koko ajan palvelimella ajettava prosessi, joka on yhteydessä web-palvelinohjelmistoon. FastCGI-tekniikkaa tukevilla web-sovelluskehyksellä ja web-palvelinohjelmistolla saataisiin aikaan paljon tehokkaampi kokonaisuus. Ohjelman web-käyttöliittymä täytyisi todennäköisesti toteuttaa uudelleen ja samalla kannattaisi muutamaa funktiota paketista *backend* optimoida. Esimerkiksi moduulissa *tokenizer* operaattorien tekstialkiot sisältävän tiedoston voisi lukea vain kerran ohjelman käynnistyksessä. Toinen huomionarvoinen seikka on, että tehtävien määrittelytiedostojen luvun yhteydessä tehdään aina samat tarkastukset. Ohjelma voisi pitää kirjaa milloin tiedostot on tarkastettu ja luvun yhteydessä tarkastaa onko tiedostoa muokattu sen jälkeen. Jos ohjelman toimintaa

täytyisi vielä nopeuttaa suurilla käyttäjämäärillä, täytyy myös itse tarkastamista optimoida. Kuten luvussa 5 useaan otteeseen todettiin, vastauksen jäsentämistä kokonaisuutena voisi optimoida.

Monimutkaisissa usean muuttujan tehtävissä vastauksen jäsentäminen ei kuitenkaan ole ohjelman nopeuskriittisin osa. Jäsentämisen osuus tarkastukseen kuluvasta ajasta on hyvin pieni verrattuna aikaan, joka kuluu vastauksen sisäisten sekä vastauksen ja oikean vastauksen välisten relaatioiden tarkastamisessa. Relaatiotarkastusten lukumäärä tehtävää kohti riippuu oikean vastauksen ja käyttäjän vastauksen pituudesta. Käytännössä se on siis tyypillisesti rajoitettu noin kymmeneen ja pahimmillaan noin sataan.

Kuten aliluvussa 3.1 todettiin, relaatiotarkastuksen ajankulutus on kertaluokassa $O(n^s)$, missä n on jokaisen muuttujan testiarvojen lukumäärä ja s on muuttujien lukumäärä. Tämän takia on ilmeistä, että tämä osa ohjelmasta tulisi optimoida mahdollisimman hyvin, jos halutaan tukea tehtäviä, joissa on suuria määriä muuttujia. Käytännössä se tarkoittaisi lausekkeiden evaluoinnin optimointia. Kannattavin evaluoinnin optimointi olisi todennäköisesti siirtyminen C++:n käyttöön. Myös aliluvussa 3.3 esitettyjen vaihtoehtojen tapaisia menetelmiä voisi mahdollisesti käyttää pienentämään evaluointien kokonaismäärän tietyissä tehtävissä jopa puoleen. Tällä saataisiin tuntuva parannus tietyissä tehtävissä, mutta muuttujien lukumäärän kasvaessa ajankulutus kasvaisi silti huimaa vauhtia.

Aivan toisenlainen lähestymistapa ongelmaan olisi luopua kaikkien testiarvojen kombinaatioiden käytöstä. Sen sijaan pyrittäisiin valitsemaan vain hyvät testiarvojen kombinaatiot, jolloin olisi mahdollista päästä eroon eksponentiaalisesta ajankulutuksesta muuttujien lukumäärän kasvaessa. Tällöin täytyisi testiarvojen valinta miettiä ja toteuttaa uudelleen ja tuloksena olisi todennäköisesti merkittävästi monimutkaisempi järjestelmä.

8. YHTEENVETO

Tässä diplomityössä määritellään kaksi matemaattista tehtävätyyppiä, jotka ovat hyvin yleisiä lukiotason harjoitustehtävissä, ja tarkastellaan toteutettua web-sovellusta, joka esittää ja tarkastaa niitä. Tehtävien vastauksien syntaksi perustuu päättelyketjuihin, minkä ansiosta käyttäjä voi sisällyttää vastaukseen välivaiheita, jotka ohjelma tarkastaa. Syntaksi on suunniteltu myös muistuttamaan mahdollisimman paljon perinteistä paperilla laskemista.

Toteutettu ohjelma päätettiin tehdä web-sovelluksena samalla tavalla kuin monet muutkin matematiikan CAA-ohjelmat, kuten STACK, AIM ja Maple TA. Web-sovelluksena ohjelmat ovat helppoja ottaa käyttöön ja opiskelijat voivat käyttää ohjelmia omilta koneiltaankin milloin vain. Web-käyttöliittymän matemaatisten lausekkeiden syntaksi on myös lähellä mainittujen ohjelmien syntakseja. Kun paperilla laskemisen merkintätapoja ei voitu noudattaa, otettiin mallia suosituista matematiikan ohjelmista, kuten Matlab ja Maple, joka toimiikin ohjelmien AIM ja Maple TA pohjana. Toteutetun ohjelman web-käyttöliittymä on kuitenkin vielä hyvin vajaa verrattuna mainittuihin ohjelmiin. Käyttöliittymästä puuttuu olennaisia toimintoja, kuten esimerkiksi tehtävien laatiminen suoraan web-käyttöliittymässä, mikä puolestaan vaatii sovellukseen sisäänkirjautumisen eri rooleissa [11].

Yksinkertaisempaa tehtävätyyppiä kutsutaan laskutehtäväksi ja sen vastaus muodostuu numeerisista lausekkeista yhtäsuuruusmerkein erotettuna. Toista tehtävätyyppiä puolestaan kutsutaan yhtälönratkaisutehtäväksi ja sen vastaus koostuu loogisista lausekkeista yhtäpitävyysmerkein erotettuna. Yhtälönratkaisutehtävät voivat sisältää myös epäyhtälöitä ja mahdollistavat päättelyketjun haarauttamisen ehtolohkoiksi kutsutulla rakenteella. Kummastakin tehtävätyypistä on kaksi variaatiota. Määritellyt tehtävät ovat erityisesti ohjelmaa varten laadittuja tehtäviä, joiden tarkastuksessa ohjelmalla on käytössään lisäinformaatiota apuna. Vapaissa tehtävissä ohjelma puolestaan pyrkii parhaansa mukaan tarkastamaan käyttäjän syöttämän vapaamuotoisen päättelyketjun.

Tehtävätyyppien puolesta mainitut matematiikan CAA-ohjelmat ovat monipuolisempia, mutta kirjoittajan tiedossa ei ole toista ohjelmaa, joka tarkastaa kokonaisen laskutoimituksen samalla tavalla kuin toteutettu ohjelma. Edellä mainituissa ohjelmissa suurin osa tehtävätyypeistä perustuu siihen, että käyttäjä syöttää ohjelmaan vain tehtävän lopullisen vastauksen. Esimerkiksi Aalto-yliopistossa käytössä oleva

STACK mahdollistaa usean vastauskentän käytön, jolloin osaa niistä voi käyttää välituloksiin, mutta se soveltuu lopulta huonosti siihen, koska käyttäjä ei voi itse määrätä niiden lukumäärää [21]. Tämä johtunee osaksi siitä, että yksi mainittujen ohjelmien tärkeimpiä käyttötarkoituksia on automaattinen arviointi. Oikeiden välivaiheiden mukaan pisteytys olisi todennäköisesti hyvin vaikeaa koneellisesti, jolloin välivaiheet eivät ole niin tärkeitä näiden ohjelmien kannalta. Toteutetun ohjelman on puolestaan tarkoitus olla nimenomaan opiskelun apuväline eikä arvioinnin työkalu.

Välivaiheiden mukaan ottamisen huonona puolena on, että vastausten syntaksi ei ole aivan yksinkertainen. Hieman samantyyppinen notaatio on kylläkin olemassa nimeltään *rakenteiset päättelyketjut* [6, 7], jota on kokeiltukin opetuksessa [8, 9, 10]. Rakenteisten päättelyketjujen käytöstä on saatu positiivisia tuloksia käytännössä, joten samantyyppisen syntaksin noudattaminen tämän ohjelman käytössä tuskin on suuri ongelma käytännössä.

Mainituista matematiikan CAA-ohjelmista poiketen toteutettu ohjelma tarkastaa vastaukset kokeilemalla vastauksen muuttujille testiarvoja. Tarkemmin ottaen jokaiselle muuttujalle määrätään testiarvot, joista muodostetaan kaikki mahdolliset kombinaatiot, joilla vastauksen osien pitävyyttä testataan. Menetelmä on yksinkertainen ja tuottaa käyttäjälle hyödyllistä palautetta, kun testiarvot, joilla virhe löytyi, annetaan käyttäjälle. Se ei kuitenkaan ole varma menetelmä kaikissa tapauksissa. Testauksessa todettiin kuitenkin, että ohjelma löytää käytännössä suurimman osan virheistä määriteltyjen tehtävien vastauksista. Yhtälönratkaisutehtävissä on erityisen tärkeää, että tiedetään tehtävän oikea ratkaisu, jotta voidaan valita hyvät testiarvot. Ilman hyviä testiarvoja ohjelma ei huomaa vastauksesta puuttuvia yhtälön ratkaisuja. Vapaissa yhtälönratkaisutehtävissä ohjelma ei tiedä oikeata ratkaisua, minkä takia ne ovat kaikkein hankalin tehtävätyyppi ja niiden tarkastus onkin melko epävarma. Laskutehtävissä puolestaan testiarvojen valinnalla ei ole suurta merkitystä ja määritellyissä yhtälönratkaisutehtävissä oikea ratkaisu onkin tiedossa.

Edellä mainitut matematiikan CAA-ohjelmat perustuvat kaikki symbolisen laskentaan. Symbolisen laskennan huono puoli on, että se on hyvin monimutkaista, mutta sillä on monia hyviä puolia verrattuna testiarvoilla kokeilemiseen. Ensinnäkin symbolisella laskennalla tarkastuksesta saadaan varma. Tarkastuksen varmuus on ehdottoman tärkeää näille ohjelmille, koska niitä käytetään arvostelun apuvälineenä. Koska toteutettu ohjelma on tarkoitettu opiskelun apuvälineeksi, tarkastuksen epävarmuus ei ole suuri ongelma, kunhan käyttäjille tehdään selväksi, että ohjelma ei löydä kaikkia virheitä. Käytännössä riittää, että se löytää suurimman osan virheistä ja tärkeämpää onkin, että ohjelma antaa käyttäjälle hyvää palautetta, joka auttaa löytämään ja ymmärtämään virheen vastauksessa.

Symbolinen laskenta antaa myös mahdollisuuden monipuolisempiin tehtäviin. Esimerkkinä STACK antaa mahdollisuuden satunnaistaa tehtäviä muun muassa va-

litsemalla satunnaiset luvut vakioiden paikoille. Symbolisella laskennalla ohjelma pystyy itse ratkaisemaan yhtälön ja vertaamaan käyttäjän vastausta siihen. Samanlainen satunnaistaminen ei onnistu toteutetun ohjelman toimintaperiaatteella, koska yhtälönratkaisutehtävien kattavaan tarkastukseen sen täytyy tietää ratkaisu ennalta. Myös satunnaistaminen palvelee automaattista arviointia, koska se estää oikeiden vastausten suoran kopioinnin ja näin ehkäisee vilppiä, kun automaattista arviointia käytetään valvomattomissa oloissa [12].

Ohjelma päätettiin toteuttaa Python-ohjelmointikielellä ja CGI-tekniikalla [13]. Tämän yhdistelmän katsottiin sopivan hyvin tällaiseen melko pieneen ja prototyyppimaiseen ohjelmaan. Toteutuksessa ohjelma jaettiin vastausten tarkastuksen hoitavaan osaan ja web-käyttöliittymään. Vastausten jäsennys ja tarkastus oli odotetusti suurempi ja hankalampi osa toteuttaa. Vastausten jäsentäjä toteutettiin itse, koska syntaksivirheiden käsittely haluttiin pitää omassa hallinnassa. Syntaksivirheiden selkeään esittämiseen pyrittiin kiinnittämään erityistä huomiota, sillä se on ohjelman käytettävyyden kannalta kriittinen ominaisuus.

Jo ohjelman suunnittelussa tuli ilmi, että sen nopeuskriittisin osa on lausekkeiden arvojen toistuva evaluointi tarkastuksen yhteydessä. Tämä pyrittiin ottamaan huomioon toteutuksessa optimoimalla evaluointia koskevaa koodia, mutta ongelma on perustavampaa laatua. Evaluointien määrä nimittäin kasvaa eksponentiaalisesti tehtävän muuttujien lukumärän suhteen. Käytännössä tämä ei kuitenkaan ole ongelma, koska tyypillinen lukiomatematiikan tehtävä sisältää vain yhden tai kaksi muuttujaa, mikä tarkoittaa, että ohjelma selviää tarkastuksesta reilusti alle sekunnissa. Symboliseen laskentaan perustuvat ohjelmat puolestaan eivät kärsi samasta ongelmasta.

Ohjelman jatkokehityksen kannalta olisi erittäin tärkeää saada se varsinaiseen koekäyttöön. Vaikka ohjelma saatiin toteutettua kuten se suunniteltiinkin ja se on testattu, pitää vielä tutkia soveltuuko se käyttötarkoitukseensa. Ennen kaikkea pitää selvittää kokevatko käyttäjät virheellisen välivaiheen osoittamisen ja esimerkkiarvojen antamisen niin hyödylliseksi, että vastauksen kirjoittaminen ohjelmaan on vaivan arvoista. Myös käyttöliittymän kehitystarvetta kannattaisi kartoittaa, jotta sitä lähdetään kehittämään oikeaan suuntaan. Teoreettisella puolella mielenkiintoinen kysymys on, että millaisia lasku- ja yhtälönratkaisutehtäviä voidaan tarkastaa varmasti kokeilemalla testiarvoja. Tässä diplomityössä esitettiin tulos vain polynomien laskutehtäville. Projekti koettiin lopulta onnistuneeksi, sillä ohjelma tukee kaikkia ominaisuuksia, joita alussa lähdettiinkin tavoittelemaan.

LÄHTEET

[1] Kankaanranta, M., Vahtivuori-Hänninen, S. & Koskinen, J. Opetusteknologia koulun arjessa - ensituloksia. Teoksessa: Kankaanranta, M. Opetusteknologia koulun arjessa. Jyväskylä, 2011, Jyväskylän yliopisto, Koulutuksen tutkimuslaitos. s. 7–13.

[2] Sangwin, C. Assessing elementary algebra with STACK. International Journal of Mathematical Education in Science and Technology, 38(2008)8. pp. 987–1002.

[3] Delius, G. Conservative approach to computerised marking of mathematics assignments. MSOR Connections 4(2004)3. pp. 42–37.

[4] Heck, A. Assessment with MapleTA: creation of test items. AMSTEL Institute: Universiteit van Amsterdam. 2004. [viitattu 26.8.2012]. Saatavissa: http://www.adeptscience.co.uk/products/mathsim/mapleta/MapleTA_whitepaper.pdf.

[5] Kangasaho, J., Mäkinen, J., Oikkonen, J., Paasonen, J., Salmela, M. & Tahvanainen, J. Pitkä matematiikka 2: Polynomifunktiot, 1. painos. Porvoo, 2004, WSOY. 177 s.

[6] Back, R., Grundy, J. & von Wright, J. Structured calculational proofs. Formal Aspects of Computing 9(1997)5. pp. 469–483.

[7] Back, R. & von Wright, J. Refinement Calculus: A Systematic Introduction. 1998, Springer-Verlag. 519 p.

[8] Back, R., Peltomäki, M., Salakoski, T. & von Wright, J. Structured Derivations Supporting High-School Mathematics. 20th Annual Symposium of the Finnish Mathematics and Science Education Research Association, Helsinki, Finland, 10.–11.10.2003. Helsinki, 2004, University of Helsinki, Department of Applied Sciences of Education. Research Report 253. pp. 104–122.

[9] Back, R., Mannila, L., Peltomäki, M. & Sibelius, P. Structured Derivations: A Logic Based Approach to Teaching Mathematics. FORMED 2008: Formal Methods in Computer Science Education, Budapest, Hungary, 29.3.–6.4.2008. Electronic Notes in Theoretical Computer Science, Elsevier Science. 10 p. [viitattu 11.7.2012]. Saatavissa: http://hera.cs.abo.fi/%7Elgrandel/formed08_final.pdf.

[10] Sallasmaa, P., Mannila, L., Peltomäki, M., Salakoski, T., Salmela, P. & Back, R. Haasteet ja mahdollisuudet tietokonetuetussa matematiikan opetuksessa. Teoksessa: Kankaanranta, M. Opetusteknologia koulun arjessa. Jyväskylä, 2011, Jyväskylän yliopisto, Koulutuksen tutkimuslaitos. s. 125–140.

[11] Rasila, A., Harjula, M. & Zenger, K. Automatic assessment of mathematics exercises: Experiences and future prospects. ReflekTori 2007 Symposium of Engineering Education, Helsinki, Finland, 3.–4.12.2007. Espoo, 2007, Helsinki University of Technology, Teaching and Learning Development Unit. pp. 70–80.

[12] Sangwin, C. Assessing higher mathematical skills using computer algebra marking through AIM. Engineering Mathematics and Applications Conference EMAC03, Sydney, Australia, 9.–11.7.2003. Australia, 2003, Engineering Mathematics Group, ANZIAM. pp. 229–234.

[13] Connolly, D. CGI: Common Gateway Interface. W3C. [viitattu 18.8.2012]. Päivitetty 12.5.2011. Saatavissa: http://www.w3.org/CGI/.

[14] Rossum, G. What is Python? Executive Summary [WWW]. [viitattu 18.8.2012]. Saatavissa: http://www.python.org/doc/essays/blurb/.

[15] Johnson, S. Yacc: Yet Another Compiler-Compiler. AT&T Bell Laboratories, Murray Hill, New Jersey 07974. [viitattu 18.8.2012]. Saatavissa: http://dinosaur.compilertools.net/yacc/.

[16] Cormen, T., Leiserson, C., Rivest, R. & Stein, C. Introduction to Algorithms, second edition. Cambridge, Massachusetts, 2001, McGraw-Hill. 1180 p.

[17] Hoare, C. Report on the Elliott ALGOL translator. The Computer Journal 5(1962)2. pp. 127–129.

[18] Norvell, T. Parsing Expressions by Recursive Descent. 1999. [viitattu 18.8.2012]. Saatavissa: http://www.engr.mun.ca/~theo/Misc/exp_parsing.htm#RDP.

[19] Walsh, N. An Introduction to Cascading Style Sheets. World Wide Web Journal 2(1997)1. pp. 147–156.

[20] Brown, R. FastCGI: A High-Performance Gateway Interface. Position paper for the workshop "Programming the Web - a search for APIs", Fifth International World Wide Web Conference, Paris, France, 6.5.1996. Open Market, Inc, 1996, Cambridge, MA. [viitattu 18.8.2012]. Saatavissa: http://www.fastcgi.com/drupal/node/6?q=node/23.

[21] Ruokokoski, J. Automatic Assessment in University-level Mathematics. Diplomityö. Helsinki 2009. Teknillinen korkeakoulu, Teknillisen fysiikan laitos. [viitattu 27.8.2012]. Saatavissa: http://users.tkk.fi/jruokoko/dipl.pdf.