T A R T U Ü L I K O O L
MATEMAATIKA-INFORMAATIKATEADUSKOND
Arvutiteaduse instituut
Informaatika eriala

Jaanus Jaggo

JavaScripti staatilise koodianalüsaatori loomine teenusena

Bakalaureusetöö

(6 eap)

Juhendaja: Siim Karus, PhD

Kaasjuhendaja: Sven Laur, PhD

Autor: …………………………………… “…..“ mai 2013

Juhendaja: ………………………………. “…..“ mai 2013

Kaasjuhendaja: …………………………. “…..“ mai 2013

Lubada kaitsmisele

Professor: ……………………..……… “…..“ mai 2013

TARTU 2013

Sisukord

Sissejuhatus 4
1 Staatiline koodianalüüs 5
1.1 Staatilise analüüsi probleemid 6
1.2 Staatilise analüüsi tehnikad 6
1.3 JavaScript 7
1.4 Teised staatilised koodianalüsaatorid 8
2 Parseri loomine 9
2.1 Parseri otstarve 9
2.2 ANTLR 10
2.3 Parsimismeetodid 11
2.4 Teised parserigeneraatorid 12
3 JavaScripti grammatika 13
3.1 Süntaksi kirjeldamise formalismid 13
3.2 JavaScripti grammatika kirjeldus 14
3.3 Grammatika esitamine ANTLR-is 14
3.3.1 Skanneri grammatika 14
3.3.2 Parseri grammatika 15
3.4 Grammatika täiendamine AST loomiseks 15
3.5 JavaScripti analüsaatori AST struktuur 17
4 AST transleerimine XML-i 19
4.1 XML 19
4.2 XPath 19
4.3 XSLT 20
4.4 Semantiline analüüs 20
5 Teenuse loomine 22

5.1 Teenuse kontseptsioon ... 22
5.2 Privaatse lähtekoodi printsiip ... 22
5.3 Teenusena loodud koodianalüsaatori eelised ... 23
5.4 Teenusena loodud koodianalüsaatori arhitektuur ... 23
6 Vigade tuvastamine ... 25
6.1 Süntaksivead ... 25
6.2 Meetrikad ... 26
6.3 Tsüklomaatiline keerukus ... 26
6.4 Hargnemiste sügavus ... 27
6.5 Tsüklite sügavus ... 27
6.6 Halvad praktikad ... 28
6.7 Näide JavaScripti analüsaatori kasutamisest ... 29
7 Järeldused ... 32
Kokkuvõte ... 33
Summary ... 34
Viited ... 35
Lisa 1. Mõisted ... 40
Kontekstivaba grammatika ... 40
Kanooniline derivatsioon ... 40
Derivatsiooni puu ... 41
Grammatika ühesus ... 41
Lisa 2. Sorteerimisprogrammi AST ... 42
Lisa 3. DVD bakalaureusetöö juurde kuuluvate failidega ... 45

Sissejuhatus

Staatiline koodianalüüs on analüüsimeetod, mille eesmärk on programmi koodi põhjal ennustada selle tööd ja leida vigu, ilma seda programmi käivitamata. Staatilised analüsaatorid leiavad peamiselt kasutust tarkvaratööstuses, kuna nad aitavad leida inimlikke vigu ning hoiavad seeläbi kokku arendajate aega. Enamus praegu kasutusel olevatest koodianalüsaatoritest on eraldiseisvad rakendused, mille kasutaja laeb internetist alla ning paigaldab oma masinasse. Niimoodi loodud tööriistad hindavad koodi kvaliteeti kindla komplekti kontrollide põhjal. Need kontrollid küll osaliselt kattuvad, kuid ei leidu sellist, mis leiaks üles kõik defektid ja võiks täielikult asendada teisi. Seetõttu peavad arendajad parima tulemuse saamiseks kasutama mitmeid erinevaid tööriistu.

Käesoleva bakalaureusetöö raames on loodud staatilise koodianalüsaatori prototüüp JavaScriptile. Selle realiseerimisel on kasutatud teenusepõhist lähenemist, kuid seda nii, et lähtekoodi ennast ei saadeta serverisse vaid analüüsitakse kliendi poolel. Sellise lähenemise eesmärk on säilitada lähtekoodi salastatus, pakkudes samal ajal võimalikult ajakohaseid kontrolle ja analüütikat. Töö eesmärgid on järgmised:

1. Anda ülevaade staatilistest koodianalüüsidest keskendudes JavaScriptile.
2. Demonstreerida JavaScripti staatilise analüüsi teostatavust teenusena.
3. Hinnata teenusena loodud analüsaatori otstarbekust.

Esimeses peatükis vaadeldakse staatilist koodianalüüsi ning tuuakse välja erinevad meetodid, mida saab selleks kasutada. Teises peatükis kirjeldatakse parseri otstarvet staatilises koodianalüsaatoris ning pakutakse välja erinevaid meetodeid selle loomiseks. Kolmandas peatükis antakse ülevaade grammatikate kirjeldamisest, seejuures vaadeldakse lähemalt JavaScripti grammatikat ning selle täiendamist abstraktse süntaksi puu loomiseks. Neljandas peatükis käsitletakse analüsaatori ülesehitust ning transleerimisalgoritmi. Viiendas peatükis kirjeldatakse teenusepõhise lähenemise eeliseid. Kuuendas peatükis vaadeldakse erinevaid vigu ning otsitakse vahendeid nende tuvastamiseks. Seitsmendas peatükis tuuakse välja järeldused. Lõpetuseks võetakse käesolev töö kokku ning pakutakse välja võimalikke tulevikuarendusi.

Tööle on lisatud formaalsete grammatikate kirjeldamiseks vajalikud mõisted (lisa 1), kuuendas peatükis näidiseks toodud sorteerimisalgoritmi abstraktne süntaksi puu (lisa 2) ja DVD plaat töö raames loodud JavaScripti analüsaatoriga (lisa 3).

1 Staatiline koodianalüüs

Et olla konkurentsivõimelised, on vaja tarkvaratootjatel hoida kõrget tarkvara kvaliteeti ja tugevat disaini. Kliendid ootavad, et tellitud tarkvara oleks defektideta ja see vastaks nende vajadustele ka paljude aastate pärast. Kuna iga tarkvara vajab hooldust, on ainult hästi struktureeritud kood pikemas perspektiivis jätkusuutlik.

Kvaliteetse tarkvara aluseks on kvaliteetne kood, mis töötab nii nagu ettenähtud, ei sisalda suuri vigu ning on loetav ja hallatav [**1**]. Ebakvaliteetne kood muudab programmeerimise aeganõudvaks ning seetõttu jääb arendajatele vähem aega uue funktsionaalsuse loomiseks. Probleemi leevendamiseks on välja töötatud erinevaid arendusmeetodeid, mis tõstavad koodi kvaliteeti ning aitavad leida programmeerimisvigu võimalikult varajases staadiumis. Üheks selliseks meetodiks on koodiülevaatused, mis tähendab lähtekoodi süstemaatilist lugemist mõne teise arendaja poolt. Koodiülevaatused on osutunud väga heaks vahendiks vigade varajaseks tuvastamiseks ja koodi kvaliteedi parandamiseks. Selle meetodi edukust kinnitavad juba IBMi 1976. aasta uuringud [**2**].

Ettevõtete jaoks on koodiülevaatused üsna kulukad, sest nende ettevalmistamine ja läbiviimine nõuab palju arendajate aega. Seepärast töötatakse alternatiivselt välja ka automatiseeritud vahendeid koodikvaliteedi tagamiseks. Need ei asenda küll täielikult koodiülevaatusi, sest nad ei tunne süsteemi konteksti ega võimalda arendajatel teadmisi ja kogemusi vahetada. Samas saab automatiseeritud analüsaatoreid kasutada edukalt koodiülevaatuste toetamiseks ning seeläbi vähendada ülevaatustele kuluvat aega.

Automatiseeritud programmi analüüsil on kaks peamist suunda:

1. Staatiline analüüs, mis tähendab programmi töö hindamist tema koodi põhjal ilma seda programmi käivitamata.
2. Dünaamiline analüüs, mis tähendab programmi töö hindamist selle töötamise ajal, erinevate sisendandmete korral.

Need analüüsimeetodid on küllaltki erinevad, sest kui staatiline analüüs sarnaneb koodiülevaatusele, siis dünaamiline analüüs sarnaneb pigem programmi testimisele. Siiski saab tarkvaraarenduse vahendites neid meetodeid edukalt omavahel kombineerida [**3**]. Käesolevas töös on keskendutud staatilisele analüüsile, mille probleeme ja tehnikaid vaadeldakse lähemalt.

1.1 Staatilise analüüsi probleemid

Kuigi staatilised analüsaatorid aitavad arendaja ja testija aega kokku hoida, on vaja nende kasutamiseks siiski arendaja abi, sest peaaegu kõik staatilised analüsaatorid raporteerivad valepositiivseid ja valenegatiivseid vigu.

- Valepositiivne (*false positive*) – analüsaator raporteerib vea, mida tegelikult programmis ei ole.
- Valenegatiivne (*false negative*) – programmis on viga, mida analüsaator peaks tuvastama, kuid seda ei tee.

Näiliselt tekitavad valepositiivsed vead arendajatele rohkem probleeme, sest võib kuluda tükk aega, enne kui arendaja saab aru, et viga tegelikult ei olegi. Sellegi poolest on valenegatiivsed ohtlikumad, sest võivad tekitada tunde, et tarkvara töötab korrektselt. Näiteks liigitasid eBay arendajad esialgu kõik staatilised analüsaatorid kasututeks, sest need ei leidnud nende jaoks mõistlikke vigu [**4**].

Staatilisi analüsaatorite puhul mõõdetakse mõistlikkust (*soundness*), terviklikkust (*completeness*) ja kasulikkust (*usefulness*). Analüsaator on „mõistlik“, kui ta ei jäta ühtegi viga tuvastamata, „terviklik“, kui ta leiab ainult päris vigu ning „kasulik“, kui tema poolt leitud vigadest on kellelegi kasu [**5**]. Enamasti tuleb nende vahel teha kompromiss, sest kui analüsaator leiab kõik vead, siis ilmselt on nende hulgas ka palju valepositiivseid ning kui ta raporteerib ainult õigeid vigu, siis jäävad paljud olulised vead tuvastamata. Hea staatiline analüsaator on selline, mis leiab kõik arendajate jaoks kasulikud vead, kuid sealhulgas raporteerib ka mõned valepositiivsed vead [**6**].

1.2 Staatilise analüüsi tehnikad

Koodianalüsaatorite taksonoomias on nimetatud neli staatilise analüüsi tehnikat [**7**]:

1. Süntaktiline mustrite sobitamine (*syntactic pattern matching*) – sobitatakse programmi koodi eelnevalt defineeritud mustrite vastu. Need mustrid võivad kirjeldada erinevaid vigu, või arvutada koodimeetrikate väärtusi.
2. Andmevoo analüüs (*data flow analysis*) – analüüsitakse programmi andmevoo graafi (*data flow graph*). Selle abil saab näiteks seostada muutujate nimesid nende varasemate väärtustamistega ja seeläbi tuvastada ületäitumisest ja nullviitadest põhjustatud probleeme.

3. Teoreemtõestus (*theorem proving*) – tõestatakse matemaatiliselt, et programm vastab oma spetsifikatsioonile. Selleks täiendatakse programmi koodi märgenditega, mis kirjeldavad formaalselt programmi tööd ning analüüsitakse nende märgendite vastavust programmi koodile.
4. Mudelkontroll (*model checking*) – koostatakse programmi lõplik olekute mudel ning hinnatakse selle põhjal programmi käitumist.

Neist tehnikatest on kõige levinum süntaktiline mustrite sobitamine, sest võrreldes teiste tehnikatega on mustrite sobitamisel põhinevaid kontrolle lihtsam realiseerida. Samas ei sobi see meetod hästi keerulisemate vigade leidmiseks ning raporteerib ka võrdlemisi palju valepositiivseid ja valenegatiivseid vigu [**8**].

Et teenusepõhisest disainist tulevaid eeliseid ära kasutada, on JavaScripti analüsaatoris oluliseks määrajaks kontrollide lisamise ja muutmise lihtsus. Seetõttu on analüüsimeetodiks kasutatud süntaktilist mustrite sobitamist. Tulemuseks saadakse nii konkreetsed veakohad kui ka kvaliteedimõõdikud, mis annavad hinnangu lähtekoodi disainile.

1.3 JavaScript

JavaScript on objektorienteeritud skriptimiskeel, mis loodi 1995 aastal Brendan Eich'i poolt Netscape's ning selle eesmärgiks oli laiendada veebilehti kliendi poolel käivitatava koodiga. JavaScripti iseloomustavad dünaamiliselt tüübitud muutujad ning prototüübil põhinev pärilus [**9**]. Dünaamilise tüüpimise tõttu tuletatakse muutujate tüübid alles programmi käivitamise ajal. Samuti võivad muutujate tüübid töö käigus muutuda, näiteks sõne tüüpi muutujale võib hiljem omistada täisarvu. Selline lähenemine teeb JavaScriptis programmide kirjutamise küll mõnevõrra mugavamaks, kuid suurendab ka tüübiteisendustest tulenevate vigade esinemise tõenäosust, sest kompilaator neid vaikimisi ei kontrolli.

Enamus tänapäevaseid veebilehitsejaid, nende seas näiteks Mozilla Firefox [**10**] ja Google Chrome [**11**], kasutavad JavaScripti käivitamiseks JIT (*just-in-time*) kompileerimismeetodit, mis tähendab, et JavaScript kompileeritakse masinkoodi täpselt siis, kui seda on vaja. Võrreldes interpreteerimisega on JIT eeliseks see, et ta on koodi korduvkäivitamisel tunduvalt kiirem. Sellegi poolest toimub JIT kompileerimine, erinevalt traditsioonilistest programmeerimiskeeltest, alles kasutaja arvutis. Üheltpoolt muudab see programmi testimise mugavamaks, kuna jääb ära aeganõudev kompileerimisprotsess. Teisest küljest on arendaja poolse kompileerimise eeliseks see, et kompilaator leiab üles kõik süntaksi-

vead, mida aga JIT-i puhul ei tehta. Seetõttu kasutavad JavaScripti arendajad erinevaid tööriistu, mis aitavad neil ka süntaksivigu leida ning staatilise analüsaatori kasutamine on üheks võimaluseks.

1.4 Teised staatilised koodianalüsaatorid

JavaScripti jaoks on tehtud mitmeid koodianalüsaatoreid, millest on järgnevalt tutvustatud kolme.

JSLint on staatiline koodianalüsaator, mis on kirjutatud JavaScriptis ja mis otsib JavaScripti lähtekoodist peamiselt stiili- ja vormistusvigu. JSLint töötab veebirakendusena, kuid seda saab kasutada ka käsurea kaudu [**12**].

JSMeter on samuti JavaScripti rakendus, mis arvutab JavaScripti lähtekoodile erinevate kvaliteedimõõdikute väärtusi, näiteks koodiridade arvu, tsüklomaatilist keerukust (*cyclomatic complexity*) ja kommentaaride arvu [**13**]. Kuigi need kontrollid on küllaltki lihtsa loomuga, annavad siiski hea võrdlusmaterjali JavaScripti rakenduste analüüsimiseks [**14**].

PDM on avatud lähtekoodiga staatiline koodianalüsaator, mis leiab vigu ja disainiprobleeme Java, JavaScripti, XML ja XSL keeles kirjutatud lähtekoodist [**15**]. PDM-i puhul on rõhku pandud mustrite sobitamisele, kuid see pakub ka andmevoo analüüsi võimalusi. Mõlema analüüsi tehnika jaoks saab arendaja ise kontrolle defineerida [**16**].

2 Parseri loomine

2.1 Parseri otstarve

Loomulikest keeltest saame me aru seetõttu, et tunneme nende struktuuri ehk süntaksit. Keele süntaks määrab reeglid, mille põhjal me fraase kokku paneme ning fraasidest omakorda lauseid moodustame. Täpselt samamoodi määrab programmeerimiskeele süntaks, kuidas tuleb selles keeles kirjutatud programmist aru saada ning seda liigendada.

Programmeerimiskeele süntaksit kirjeldatakse enamasti kahel tasemel:

- Konkreetne süntaks (*concrete syntax*) defineerib keele grammatikareeglid ning määrab, kuidas selles keeles kirjutatud programmi liigendada. Konkreetne süntaks sisaldab palju sellist, mis on vajalik eelnevuskonfliktide vältimiseks ja keele parsitavaks muutmiseks, kuid ei oma semantilist tähendust.
- Abstraktne süntaks (*abstract syntax*) defineerib keele semantilisel (sisulisel) tasemel, vastavalt sellele, millised on elementide omavahelised seosed [**17**].

Programmi süntaktilist analüüsi nimetatakse ka parsimiseks (*parsing*). Selle käigus viiakse programm puu kujule, sest nii on seda arvutil lihtsam töödelda. Tarkvara, mis sellist teisendamist teeb, nimetatakse parseriks: “Parser on süntaksianalüsaatorit, mis liigendab programme või muud teksti, sageli assembleerimise, kompileerimise või analüüsi esimese sammuna.“ [**18**]. Kõige sagedamini kasutatakse parserit translaatorite loomiseks. Translaator on programm, mis teisendab ühes keeles (lähtekeeles) kirjutatud programmi teise keelde (objektkeelde). Transleerimine koosneb enamasti mitmest etapist, sest oskuslikult valitud etapid kiirendavad transleerimise protsessi. Raamatu „Formaalsed keeled, grammatikad ja translaatorid“ [**19**] põhjal on tüüpilised transleerimise etapid järgmised:

1. Leksikaanalüüs - selle käigus moodustatakse lähtetekstist lekseemide jada. Lekseemideks on iseseisvad programmi nähtava teksti osad: sõned, identifikaatorid, kommentaarid, reserveeritud sõnad ja eraldajad.
2. Süntaksanalüüs - selle käigus konstrueeritakse lekseemide jadast süntaksi puu ehk derivatsioonipuu.

3. Abstraktse süntaksi puu moodustamine - selle käigus eemaldatakse süntaksi puust kõik ebaoluline ja nii-öelda süntaktiline suhkur, milleks on näiteks sulud, mida on vaja küll eelnevuskonfliktide vältimiseks programmi parsimisel, kuid süntaksi puus ei oma enam sisulist tähendust. Samuti võib AST-i struktuuri võrreldes derivatsiooni puuga muuta, näiteks selleks, et edasine analüüs oleks kiirem.
4. Semantiline analüüs - selles etapis genereeritakse abstraktsest süntaksi puust objektkeelne väljundkood.

JavaScripti analüsaatoris on lähtekoodi analüüsimiseks loodud translaator, mis teisendab JavaScriptis kirjutatud programmi XML dokumendiks. See XML hoiab esialgse programmi abstraktset süntaksit, mida saab analüüsida käesoleva töö neljandas peatükis kirjeldatud vahenditega.

Tänapäeval on parserite loomine muutunud üsna levinud tegevuseks ning käsitsi parseri kirjutamise asemel on tihti otstarbekam kasutada parserigeneraatorit. Parserigeneraator on programm, mis koostab keele kirjelduse järgi sellele keele parseri. Kuna sisendiks olev keele kirjeldus on kompaktsem ja paremini loetavam kui parseri lähtekood, on parserigeneraatoriga loodud parsereid lihtsam muuta ja täiendada. Tänapäevased parserigeneraatorid on hästi optimeeritud ja koostavad kiireid parsereid, mis oskavad ka mõistlikult veateateid raporteerida [**20**].

2.2 ANTLR

ANTLR (*ANother Tool for Language Recognition*) on parserigeneraator, mille autoriks on San Francisco ülikooli professor Terence Parr. Tegemist on vabavaralise tööriistaga, mida saab kasutada interpretaatorite, kompilaatorite ja teiste keeletuvastustööriistade loomiseks. Erinevalt paljudest teistest parserigeneraatoritest toetab ANTLR mitmeid väljundkeeli, ehk temaga saab genereerida parsereid näiteks Java, C++, C#, Python või PHP keeles [**21**]. JavaScripti analüsaatori loomiseks on kasutatud ANTLR-i versiooni 3.1, sest selle loomise ajal oli see viimane versioon ANTLR-ist, mis liidestus .NET raamistikuga.

ANTLR-i abil saab genereerida järgmiseid translaatori komponente:

- Skanner (*lexer*) - teostab leksikaanalüüsi, ehk saab sisendiks lähtekoodi sümbolite voo ning genereerib sellest lekseemide (*tokens*) jada.
- Parser - saab sisendiks skanneri poolt genereeritud lekseemide jada ning koostab sellest abstraktse süntaksi puu.

- Puuläbija (*tree walker*) - saab sisendiks abstraktse süntaksipuu ning läbib selle rekursiivselt. Puuläbijat võib kasutada näiteks selleks, et valideerida abstraktse süntaksi puu semantilist struktuuri või genereerida objektkeelne väljundkood.

Nende komponentide genereerimiseks tuleb kirjeldada analüüsitava lähtekeele leksika ja grammatika ANTLR-ile sobival kujul. Selleks on ANTLR-il oma grammatikate kirjeldamise keel, mis põhineb EBNF (*Extended Backus-Naur Form*) kujul, kuid lisab sellele juurde mõned täiendused, mis näiteks aitavad kirjeldada ka abstraktse süntaksi puu struktuuri. ANTLR-is grammatikate kirjeldamist on täpsemalt käsitletud käesoleva töö kolmandas peatükis.

2.3 Parsimismeetodid

Üheks oluliseks tunnuseks, mille põhjal parserigeneraatoreid üksteisega võrreldakse, on parsimisemeetod, mida nad kasutavad. Enamikes parserigeneraatorites kasutatakse kontekstivabade grammatikate parsimiseks ühte kahest meetodist: kas ülevalt alla parsimist (*LL parsing*), või alt üles parsimist (*LR parsing*).

Ülevalt alla parsimise puhul alustab parser aksioomist ning rakendab sellele produktsioone nii, et ta jõuaks välja sisendvoost tulevate lekseemideni. Ülevalt alla parsereid liigitatakse omakorda selle järgi, mitut lekseemi nad maksimaalselt korraga vaadata saavad. Seda, mitu lekseemi parser korraga maksimaalselt vaatab, tähistatakse sulgudes oleva numbriga, näiteks LL(1) parserid näevad korraga ainult ühte lekseemi. See seab piirangu grammatikatele, mida nad parsida saavad. Näiteks kui LL(3) parser analüüsib programmi, milles ei piisa õige produktsiooni valimiseks ainult kolme lekseemi vaatamisest, vaid on vaja vaadata ka neljandat ja viiendat lekseemi, jääb see parser sellega hätta. Paraku eksisteerib ka grammatikaid, mis ei ole LL(k) grammatikad ühegi konstantse „k“ väärtuse korral. Nendes keeltes kirjutatud programmide parsimiseks on vaja teisi parsimismeetodeid [**22**].

Alt ülesse parsimise korral alustab parser sisendvoo lekseemidest, sobitades neid produktsioonide paremate pooltega. Leides sobiva produktsiooni rakendab parser seda tagurpidi, asendades need lekseemid produktsiooni vasakus pooles oleva mitteterminaliga. Samamoodi jätkates asendab ta järjest elemente, kuni aksioomini välja.

ANTLR-i kolmas versioon kasutab parsimiseks LL(*) meetodit, mis lubab parsida ka grammatikaid, millega LL(k) (fikseeritud arvu lekseeme käsitlevad) parserid hätta jääksid,

ning seeläbi saab koostada keeruliste keelte jaoks loomulikumaid grammatikaid. Sellegi poolest on võrreldes ANTLR versioon kahega, mis kasutab LL(k) parsimismeetodit, ANTLR-i kolmanda versiooniga genereeritud parserid sama grammatikat kasutades umbes 2,5 korda kiiremad [**23**].

2.4 Teised parserigeneraatorid

Lisaks ANTLR-ile on olemas teisigi parserigeneraatoreid, millest tuntumad on Lex *(Lexical Analyzer Generator)* ja Yacc *(Yet Another Compiler-Compiler).*

Yacc on Stephen C. Johnsoni loodud parserigeneraator Unixi platvormidele. Yacc kasutab LR parsimismeetodit ning genereerib parsereid C keeles. Samuti on Yacci grammatikate kirjeldamisel võetud malli C keele süntaksist [**24**]. Yacci grammatikaid kasutavad ka mitmed teised parserigeneraatorid, näiteks parserigeneraator Bison.

Lex on Mike Leski poolt loodud parserigeneraator, mis keskendub leksika analüüsile. Lex on disainitud nii, et seda saaks kasutada koos Yacci parseritega ning genereerib samuti C keelset koodi. Lexi grammatikad kasutavad sisendi töötlemiseks regulaaravaldisi, mistõttu saab sellega luua küllaltki keerukaid analüsaatoreid [**24**]. Ka Lexi põhjal on loodud teisi leksika analüsaatoreid, näiteks Flex (*Fast Lexical Analyzer*).

Translaatorite tegemise süsteem (TTS) on Ain Isotamme loodud parserigeneraator, mis on mõeldud eeskätt Tartu Ülikooli tudengitele transleerimismeetodite ja süntaktilise analüüsi õpetamise hõlbustamiseks. Oluline roll TTSi kujundamisel on ka professor Mati Tombakul, kelle meetoditel antud süsteem põhineb. TTS sisaldab nii leksika analüsaatorit (Skanner) kui ka parseri osa, ning viimane on omakorda jagatud väiksemateks eraldi etappideks, mis annavad hea ülevaate süntaksanalüüsi tööpõhimõttest [**25**].

3 JavaScripti grammatika

3.1 Süntaksi kirjeldamise formalismid

Formaalseid meetodeid kontekstivabade grammatikate kirjeldamiseks hakati välja töötama 1950ndatel aastatel. Esimese sellealase publikatsiooni [**26**] andis välja Noam Chomsky, kus ta esitas kolme mudeli teooria. Chomsky formalismi hakati nimetama Chomsky normaalkujuks (*CNF: Chomsky Normal Form*). Oma metakeele grammatikate kirjeldamiseks defineeris ka W. Backus, kes oli varasemalt tutvunud Chomsky töödega. Seda keelt arendas edasi Peter Naur, lihtsustades Bacuse tähistusi ning selle tulemusena sündis Bacus-Nauri metakeel (*BNF: Bacus-Naur Form*) [**25**], millel baseeruvad paljude tänapäevaste parserigeneraatorite grammatikad. Hiljem on Bacus-Nauri metakeelele lisatud erinevaid laiendusi, näiteks ABNF (*Augmented Bacus-Naur Form*) ja EBNF (*Extended Bacus-Naur Form*). Käesoleva töö raames on käsitletud lähemalt EBNF-i, mille võttis kasutusele programmeerimiskeele Pascal looja Niclaus Wirth. EBNF lisab BNF-le mitmeid lihtsustavaid operaatorid, millest osasid on tutvustatud tabelis 1.

Tabel 1. EBNF-i operaatorid

Operaator	Kirjeldus
?	ütleb, et produktsiooni paremas pooles olev element (või grupp elemente, kui need on sulgudes) võib esineda 0 või 1 korda.
*	ütleb, et produktsiooni paremas pooles olev element (või grupp elemente, kui need on sulgudes) võib esineda 0 kuni n korda.
+	ütleb, et produktsiooni paremas pooles olev element (või grupp elemente, kui need on sulgudes) võib esineda 1 kuni n korda.

Kuigi lisandunud operaatorid muudavad grammatikate kirjeldamist mugavamaks, ei ole EBNF tegelikult võimsam, kui BNF, sest kõik EBNF-i produktsioonid on teisendatavad BNF-i produktsioonideks [**22**]. Et EBNF-l põhinevaid grammatikaid saaks ühtselt ka arvuti lugeda, on EBNF standardiseeritud ISO/IEC 14977 standardiga [**27**], mis määrab grammatika elementide täpse tähistuse.

3.2 JavaScripti grammatika kirjeldus

JavaScript baseerub ECMAScripti standardil ning konkreetne realisatsioon on veebilehitsejate teostada. Seetõttu võivad JavaScripti programmid töötada erinevates veebilehitsejates veidi erinevalt. Näiteks Mozilla Firefox lisab JavaScriptile mõningaid lisasid, mida teised veebilehitsejad ei pruugi toetada, nende seas võtmesõnad `let` ja `yield` [**28**].

JavaScripti analüsaatori loomisel on tuginetud ECMA-262 standardi versioonile 5.1 [**29**]. See määrab grammatika leksika ja konkreetse süntaksi, kuid jätab abstraktse süntaksi parseri loojale defineerida.

3.3 Grammatika esitamine ANTLR-is

JavaScripti staatilise analüsaatori loomiseks on genereeritud ANTLR-i abil leksika analüsaator ehk skanner ja parser. Mõlema jaoks kasutab ANTLR ülevalt alla parsimismeetodit ja ühtset grammatika kirjeldamise viisi. Neid grammatikaid saab hoida koguni samas failis, mis teeb nende koostamise mugavamaks. Sellegi poolest on neil grammatikatel mõned olulised erinevused.

3.3.1 Skanneri grammatika

Kontekstivaba grammatika lekseemid on defineeritud skanneri grammatika poolt. Selle grammatika reeglid on ka lekseemide nimedeks ning peavad algama suure algustähega. Iga skanneri grammatika reegel on iseseisev ning defineerib ühe lekseemi. Paremaks esitamiseks saab neid reegleid jagada osadeks ehk fragmentideks (*fragment*), mis võimaldab reegli osasid ka teistes reeglites taaskasutada.

```
Kümnendarv
  : Number+ ’.’ Number*;
fragment Number
  : (’0’..’9’);
```

Näidis 1. Kümnendarvu esitamine ANTLR-i grammatikaga

Näidisel 1 defineeritakse fragment „Number“, milleks võib olla märk nullist üheksani. Seda kasutades esitatakse kümnendarv ühe kuni mitme järjestikuse numbrina, millele järgneb punkt ja veel null kuni mitu järjestikust numbrit. Seega selle definitsiooni järgi sobivad kümnendarvudeks „1.“, „23.5“ ja „100.9999“.

3.3.2 Parseri grammatika

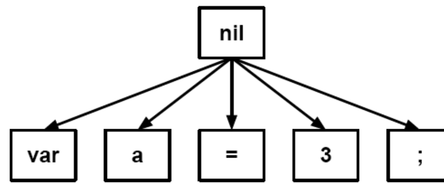
Parseri grammatika on samuti kogum reegleid, kuid erinevalt skanneri grammatikast ei ole need reeglid eraldiseisvad, vaid defineeritakse teiste kaudu. Kontekstivaba grammatika mõistes on nende reeglite näol tegemist produktsioonidega, mis määravad keele konkreetse süntaksi. Süntaksanalüüsi alustatakse algusreeglist (*start rule*), milleks on kontekstivabas grammatikas aksioom. Kõik teised reeglid saab sellest reeglist tuletada, kuni nad on jagatud terminalideks, millele vastavad skannerist tulevad lekseemid.

Sellisel viisil on kirjeldatav keele konkreetne süntaks, ehk täpsed grammatikareeglid, mille abil saab seda programmi parsida. Abstraktse süntaksi puu moodustamiseks lubab ANTLR täiendada parseri grammatikat ka abstraktse süntaksi elementidega. Seega sisalduvad nendes reeglites koguni kahe kontekstivaba grammatika elemendid (üks määrab konkreetse ja teine abstraktse süntaksi). Transleerimise kiiruse huvides on enamasti nõutud, et konkreetse süntaksi grammatika peab olema ühene. Abstraktse süntaksi grammatika puhul pole ühesust nõutud, kuid see tuleb puu hilisemal töötlemisel kasuks [**30**].

JavaScripti analüsaatori loomisel on kasutatud Chris Lambrou koostatud JavaScripti grammatikat ANTLR-ile. Seda grammatikat tohib kasutada BSD litsentsi all, mis lubab seda muuta ja levitada, kuid nõuab, et autori nimi oleks ära märgitud [**31**]. Chris Lambrou grammatika vastab peaaegu täielikult ECMAScripti versiooni 5.1 spetsifikatsioonile [**29**] ning puuduvad reeglid on JavaScripti analüsaatori jaoks kohandamisel juurde lisatud. Samuti sisaldab see nii skanneri kui parseri grammatikat, kuid jätab abstraktse süntaksi puu defineerimata.

3.4 Grammatika täiendamine AST loomiseks

Vaikimisi loob ANTLR süntaksi puu, mis kujutab endast lihtsalt lekseemide järjendit. Näiteks omistamisprogrammi `var a = 3;` parsimisel on tulemuseks joonisel 1 kujutatud süntaksi puu.



Joonis 1. Struktureerimata AST

Sellest puust pole programmi analüüsimisel eriti kasu, sest see ei väljenda programmi süntaktilist struktuuri. Et ANTLR genereeriks sobiva puu, tuleb grammatikat täiendada abstraktse süntaksiga. Selleks on ANTLR-is olemas vastavad operaatorid ja ümberstruktureerimise reeglid (*rewrite rules*). AST-i struktureerimise operaatorid on toodud tabelis 2.

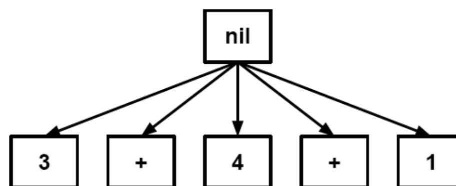
Tabel 2. ANTLR-i abstraktse süntaksi puu loomise operaatorid

Operaator	Kirjeldus
!	Selle operaatoriga märgistatud elementi AST-i ei lisata. Selle abil eemaldatakse AST-st kõik ebavajalik (nn süntaktiline suhkur).
^	Selle operaatoriga märgistatud element märgitakse uueks tipuks. Kõik ülejäänud produktsiooni paremasse poolde jäävad elemendid saavad selle tipu alluvataks.

Järgnevalt on toodud näide, mis illustreerib nende operaatorite kasutamist. Naturaalarvude liitmiseks võib defineerida ANTLR grammatika reegli:

```
liitmine : INT ('+' INT)* ;
```

Selles reeglis tähistab INT mistahes täisarvu ja tärn sulgude järel ütleb, et sulgude sees olevad elemendid võivad korduda 0 kuni n korda. Selle definitsiooni järgi vastab avaldisele *3+4+1* joonisel 2 kujutatud abstraktne süntaksi puu.

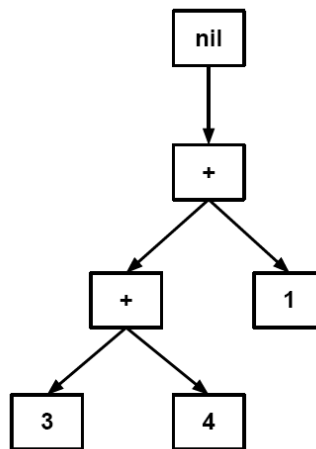


Joonis 2. Struktureerimata liitmisavaldise AST

Et tulemuseks saadud abstraktse süntaksi puu struktuur oleks sobiv, täiendame seda reeglit puu loomise operaatoriga järgnevalt:

```
expr : INT ('+'^ INT)* ;
```

Selle tulemusena märgitakse iga + sümbol uueks tipuks ning tulemuseks on joonisel 3 kujutatud puu, mille väärtuse saab välja arvutada, kui läbida see puu eesjärjestuses.



Joonis 3. Struktureeritud liitmisavaldise AST

Sellisel viisil saab määrata, millistest lähteteksti sümbolitest toimuvad hargnemised. Selleks, et AST-i semantiliselt modifitseerida ning lisada sinna ise defineeritud tippe (*tokens*), on alates ANTLR-i kolmandast versioonist võimalik kasutada puu ümberstruktureerimise reegleid (*rewrite rules*). Ümberstruktureerimise reeglid kirjutatakse grammatikasse kujul:

```
konkreetne süntaks -> abstraktne süntaks
```

Näiteks saab sel viisil lisada AST-i ise defineeritud tipu PROGRAM, millele vastav grammatikareegel on kujul:

```
Program : sourceElements? EOF -> ^(PROGRAM sourceElements?);
```

3.5 JavaScripti analüsaatori AST struktuur

Staatilise analüsaatori AST struktuuri sobivuse hindamiseks on kolm kriteeriumi: [**16**]

- Esiteks, kui hästi vastab AST selle keele mudelile, mille struktuuri see kirjeldab. Seetõttu peaks AST sisaldama üldjoontes samu elemente mis on lähtekoodiski.

- Teiseks, kas iga AST tipu juurde kuuluvad ressursid on kergesti kättesaadavad. Näiteks `for` tsüklile vastava tipu puhul võib eeldada, et selle alluvate hulgast leiduvad eeltegevused, jätkamistingimus, järeltegevused ja sisu, ilma et peaks neid eraldi puud läbides otsima.
- Kolmandaks, kas AST on ühene. Seega igale lausele (*statement*) koodis peaks vastama ka kindel konstruktsioon AST-s.

JavaScripti analüsaatoris on abstraktse süntaksi puu mitteterminalid tähistatud suurte tähtedega. AST mitteterminalideks on nii keele reservsõnadele vastavad tipud, näiteks IF, TRY, FOR kui ka lisaks defineeritud mõisted, mis aitavad kirjeldada semantilist struktuuri, näiteks ARRAY, STRING ja PROPERTY.

Kõigi operaatorite jaoks märgitakse abstraktsesse süntaksi puusse OPERATOR tipp, mille esimeseks järglaseks on vastava operaatori märk ning ülejäänud järglased on operaatori argumendid. Samuti luuakse AST-i eraldi tipud andmete tüübi tähistamiseks. JavaScriptis on nendeks algtüüpideks *String*, *Number*, *Boolean*, *Array*, *Object* ja *Null*. Näiteks sõne tähistamiseks luuakse tipp STRING ning sõne väärtus märgitakse selle tipu alamtipuks.

Selline tähistus võimaldab AST struktuuri talletada nende märkide tüübid, mis vastasel juhul läheksid kaduma. Samasuguse tähistusega arvestatakse hiljem ka süntaksi puu transleerimisel XML-i, mistõttu saab XML-i eeliseid ära kasutada. Täpsemalt on sellest kirjutatud järgmises peatükis.

4 AST transleerimine XML-i

Objektkeel, millesse lähtekood transleeritakse, tuleks valida nii, et see muudaks hilisemat analüüsi teostamise kergemaks, kuid samas säilitaks ka esialgse struktuuri. JavaScripti analüsaatori puhul on nõudeks see, et objektkeelele saaks esitada keerulisi päringuid, mille abil abstraktset süntaksi puud veamustrite vastu kontrollida. Kõige paremini sobib JavaScripti analüsaatoris objektkeeleks abstraktse süntaksi puu struktuuri kirjeldav XML dokument. XML dokumentidel põhinevat analüüsimeetodit kasutab ka staatiline analüsaator PMD.

4.1 XML

XML (*Extensible Markup Language*) on andmete hoidmiseks ja transportimiseks mõeldud märgendkeel. XML defineerib süntaksi, mida selle keele perekonda kuuluvad keeled jagavad, kuid jätab defineerimata leksika. See tähendab, et XML lubab dokumendi autoril ise vajalikud märgendeid defineerida, mida ta oma dokumentides kasutab. XML-i disainimisel on seatud eesmärgiks, et see oleks inim- ja masinloetav ning sobiks ka läbi veebi kasutamiseks [**32**].

XML baseerub programmeerimisliidesel DOM (*Document Object Model*), mis defineerib XML dokumendi loogilise struktuuri ning selle, kuidas õigesti dokumendi poole pöörduda. DOM esitab XML dokumendi objektidest koosneva puustruktuurina, mis sarnaneb objekt-orienteeritud keeles esitatud puuga. DOM puu töötlemiseks on olemas mitmeid erinevaid vahendeid, näiteks XSLT, XPath, XOM, JDOM ja teised, kuid käesoleva töö raames vaadeldakse neist lähemalt XSLT-d ja XPath-i.

4.2 XPath

XPath on päringukeel, mis on mõeldud peamiselt DOM puus elementide selekteerimiseks, kuid võimaldab teha ka erinevaid matemaatilisi arvutusi, stringitöötlust ja Boole'i algebrat. Esimene versioon XPath-st avaldati 1999. aastal, samal ajal XSLT-ga [**33**] ning seda kasutatakse XSLT mallide sees DOM puu elementide poole pöördumiseks.

4.3 XSLT

XSLT (*Extensible Stylesheet Language Transformations*) on keel, mis teisendab XML dokumentide struktuuri ja sisu. XSLT abil saab teisendada mingi XML dokumendi teiseks XML dokumendiks, muutes vastavalt DOM puu struktuuri. XSLT esimene versioon tuli välja 1999. aastal ja sai kohe üsna populaarseks. Praegune versioon 2.0 avaldati 2007. aastal ning see parandas mitmeid XSLT esimese versiooni kitsaskohti, näiteks veatöötlust ning ühilduvust teiste XML standarditega [**33**].

XSLT dokument on jaotatud reegliteks, millest igaüks määrab, kuidas teatud XML elemente tuleb töödelda. Kuna XSLT on deklaratiivne keel, siis pole selle reeglite järjekord oluline, vaid XML elemente töödeldakse selles järjekorras, milles nad dokumendis on.

XSLT on ka Turingi täielik keel [**34**], see tähendab et matemaatiliselt saab XSLT mallidega lahendada samu probleeme, mida traditsiooniliste programmeerimiskeeltegagi. Sellegi poolest on XSLT eeskätt dokumentide teisendamiseks mõeldud, kuid kuna sellega on kirjeldatavad ka keerulisemaid konstruktsioone, sobib see staatilises analüsaatoris hästi AST struktuuri kontrollimiseks.

4.4 Semantiline analüüs

Semantilise analüüsi käigus genereeritakse abstraktsest süntaksi puust XML DOM puu. Kuna me tahame lähtekoodi analüüsida originaalkeelest lähtuvalt, siis peaks ka vastav XML dokument sarnanema oma struktuurilt abstraktsele süntaksi puule.

Vaikimisi on ANTLR-i loodud parseri abstraktse süntaksi puu tüübiks „CommonTree“. Tegemist on ANTLR-i enda puustruktuuriga, mis sobib hästi parsimisoperatsioonide tegemiseks.

Ühe variandina võib teisendaja loomiseks kasutada ANTLR-i genereeritud puuläbijat. Selleks on tarvis kirjeldada puuläbija grammatika, mis sarnaneb suurel määral parseri grammatikale ning loob sellega semantilise lisakihi. Antud juhul ei ole see vajalik ning vähendab pigem analüsaatori paindlikust, sest iga muudatuse tegemiseks tuleb teha vastav muudatus ka puuläbija grammatikas.

Huvitava lahendusena on Terence Parr pakkunud välja võimaluse asendada klassikaline „CommonTree“ tüüpi abstraktse süntaksipuu tipp uue tipuga, mis päriks „CommonTree“ tippu, ning samas realiseeriks ka liidest „XDomNode“, mis on ühtlasi XML DOM puu

tipuks [**35**]. Nimelt vaatleb ANTLR sisemiselt kõiki abstraktse süntaksi puu tippe „Object“ tüüpi isenditena. Seega on abstraktse süntaksi puu tüübi vahetamiseks vaja koostada vaid vastav puu adapter. Sellisel viisil saab viia JavaScripti süntaktilise analüüsi ja objektkeelde transleerimise samaks protseduuriks. Ometi ei pruugi see parandada transleerimise kiirust, sest DOM pole parsimisoperatsioonide tegemiseks kõige parem struktuur. Näiteks on DOM puu juure muutmine kulukas, kuid parsimise käigus vahetub see tihti.

JavaScripti analüsaatoris on XML-i loomiseks koostatud translaator, mis läbib abstraktse süntaksi puu näidisel 2 kujutatud algoritmi järgi.

```
Def:teisenda(tipp)
  alustaXMLtippu(tipp.nimi)
if tipp.esimeneJärglane = veaTipp
    lisaXMLVeaTipp()
else if tipp.esimeneJärglane ∈ V_T
      lisaAtribuut(„value“, esimeneJärglane.nimi)
else
    teisenda(tipp.esimeneJärglane)
    for järglane in tipp.ülejäänudJärglased
    if tipp.esimeneJärglane = veaTipp
      lisaXMLVeaTipp()
    else
      teisenda(järglane)
lõpetaXMLtipp()
```

Näidis 2. Transleerimisalgoritm

Iga tipu kohta kontrollitakse esmalt tema esimest järglast, kui tegemist on abstraktse süntaksi puu terminaliga, siis lisatakse esimese järglase väärtus selle tipu „value“ atribuudiks. Kui mitte, siis töödeldakse teda nii nagu kõiki teisigi tippe, ehk kutsutakse uuesti välja funktsiooni „teisenda“ ning seeläbi lisatakse see uueks XML tipuks.

See meetod eeldab, et abstraktses süntaksi puus eelneb igale terminalile vahetult vähemalt üks mitteterminal ning iga terminal märgitakse talle eelneva mitteterminali „value“ atribuudiks. Näiteks terminali + vanemaks on alati tipp OPERATOR ja funktsiooni „myFunction” eellaseks on tipp IDENTIFIER. Et see tingimus kehtiks, peab ka parseri grammatika olema vastavalt koostatud.

5 Teenuse loomine

5.1 Teenuse kontseptsioon

Olemasolevad staatilised koodianalüsaatorid ei leia kõiki programmeerimisvigu, sest nad sisaldavad fikseeritud arvu kontrolle, millele vastavaid vigu nad otsivad. Keerukamad analüsaatorid, näiteks PMD ja FindBugs [**36**], võimaldavad kasutajatel endal uusi kontrolle lisada ning kasutada neid ka kontekstist lähtuvate vigade leidmiseks [**37**]. Valdkonna põhiste kontrollide lisamine nõuab arendajalt üsna palju lisatööd ja ei ole alati otstarbekas. Samuti peab arendaja väga täpselt teadma, milliseid kontrolle lisada. See teadmine tekib alles üsna pikaajalise kogemuse põhjal. Käesolevas töös pakutakse sellele probleemile lahenduseks teenusepõhist lähenemist.

Veebiteenus on mingi avalikuks tehtud protseduur, meetod või objekt koos talle juurde kuuluva liidesega, mida saab interneti vahendusel kasutada. Oma olemuselt on teenus ühe programmi suhtlemine teisega mingi toimingu tegemiseks või andmete jagamiseks [**38**]. Teenuste puhul on kaks poolt: server, kes teenust pakub, ning klient, kes seda kasutab. Teenusepõhine lähenemine võimaldab luua paindlikumaid rakendusi, sest loob vahetu kanali teenuse pakkuja ning tarbija vahel.

Teenuse kasutamine staatilise koodianalüsaatori loomisel muudab selles teostatavad kontrollid kergemini muudetavaks ning neid saab paremini lõppkasutaja vajadusele kohandada [**39**]. Seega lisab selline lähenemine mitmeid uusi võimalusi analüüsi paremaks muutmiseks, kuid samas kerkib esile ka lähtekoodi konfidentsiaalsuse küsimus: „miks peaks arendaja saatma oma kõrge intellektuaalse väärtusega koodi serverisse, kus saab sellest koopia teha ja kellelgi teisele edasi müüa?“. Selle probleemi vastu pole kaitstud ka lokaalsed arendusvahendid, sest neis võib samuti leiduda tagauksi, mille kaudu lähtekood ära varastada. Teenusepõhise lähenemise puhul kerkib see küsimus siiski palju tõsisemalt esile.

5.2 Privaatse lähtekoodi printsiip

JavaScripti analüsaatoris on kasutatud privaatse lähtekoodi printsiipi, ehk lähtekoodi ei saadeta serverisse, vaid analüüs viiakse läbi kliendi poolel. Server saadab selleks kliendile

vastavad juhised, mis on kirjeldatud XSLT mallidena. Nende mallide põhjal teostatakse kliendi poolel DOM dokumendina esitatud abstraktse süntaksi puu teisendused ning saadud tulemused saadetakse edasiseks töötluseks tagasi serverisse.

Sellisel viisil ei saada arendaja oma lähtekoodi küll serverisse, kuid ometi saab teenuse poolel koguda vajalikku tagasisidet, mille abil analüüsiprotsesse tõhusamaks muuta. Viimase sammuna võib server saata kliendile ka analüüsi kokkuvõtte, mis sisaldaks koondinfot analüüsi tulemuste kohta, kuid see jääb käesoleva töö skoobist välja.

5.3 Teenusena loodud koodianalüsaatori eelised

Siim Karus toob oma artiklis „XML Development with Plug-Ins as a Service“ [**40**] välja mõned eelised, mille andis XML failidest vigade leidmise teegi XMLStyleHelper muutmine teenuseks. JavaScripti analüsaatori seisukohalt on neist olulised järgmised:

1. Teenust on lihtsam uuendada, kui eraldiseisvat rakendust. Kuna XSLT kujul kontrolle saab hoiustada ka andmebaasis, siis tähendab nende muutmine vaid vastava andmebaasipäringu tegemist.
2. Teenuse abil saab jagada analüsaatori konfiguratsiooni erinevate klientide vahel, mistõttu on kõigil kasutajatel juurdepääs kõige värskemale infole.
3. Teenuse poolel saab koguda informatsiooni teenuse kasutajate analüüsi tulemuste kohta ning teha selle põhjal statistikat. Saadud infot saab kasutada selleks, et muuta analüsaatorit tõhusamaks, täiendades kontrolle ning täpsustades nende põhjal tehtavaid järeldusi.

5.4 Teenusena loodud koodianalüsaatori arhitektuur

JavaScripti analüsaator on realiseeritud laiendusena Siim Karuse XMLStyleHelper teenusele võimaldades sellel lisaks XML dokumentidele analüüsida ka JavaScripti lähtekoodi. Veebiteenuse realiseerimiseks on JavaScripti analüsaatoris kasutatud XMLStyleHelperi komponente, mis tagavad turvalise andmevahetuse serveriga.

JavaScripti analüsaatori arhitektuur on esitatud töövoona (*workflow*). Töövoog kujutab endast mitut tegevust sisaldavat protsessi, mille tulemusena jõutakse püstitatud eesmärgini [**41**].

Tööprotsesside defineerimine töövoona annab järgmised eelised:

1. Parem ülevaade tööprotsessidest – tegevuste selge piiritlemine ja järjestamine võimaldab tööprotsessi visualiseerida ning näha tervikpilti.
2. Võimalus komponente taaskasutada – kuna iga töövoo tegevus on iseseisev komponent ning omab selgeid sisendeid ja väljundeid, saab seda hõlpsasti taaskasutada.
3. Komponentide läbipaistvus – kuna töövoo tegevused suhtlevad teistega ainult kindlate sisendite ja väljundite kaudu, on vigast komponenti lihtsam tuvastada. Samuti saab luua konkreetsed testjuhtumid nende komponentide kontrollimiseks.

Microsofti .NET raamistikud on alates 2006 aastast sisaldanud töövoogude defineerimise võimalust. Vastav raamistik kannab nime Windows Workflow Foundation ja see sisaldab erinevaid töövahendeid, kaasarvatud visuaalset keskkonda, töövoogude tegemiseks ja haldamiseks [**42**]. Kuna ka XMLStyleHelperi arhitektuur tugineb Windows Workflow Foundationi töövoogudel, ühildub see JavaScripti analüsaatori komponentidega. XMLStyleHelperiga saab koostada erinevaid analüüsi kirjeldusi, mis pannakse kokku töövoogudena. Need töövood saadetakse pistikprogrammina (*Plug-in*) kliendile. Samuti saavad pistikprogrammid oma töövoo sees serveriga suhelda.

JavaScripti lähtekoodi analüüsimiseks saadab XMLStyleHelperi teenus kliendile pistikprogrammi, mis sisaldab JavaScripti analüsaatori töövoogu. See töövoog koosneb komponentidest, mida on vaja JavaScripti transleerimiseks XML-i ning selle analüüsimiseks. Lõpuks saadab XMLStyleHelper teenus serverisse analüüsi tulemuste XML dokumendi, mis talletatakse seal hilisemaks töötlemiseks ja statistika tegemiseks.

6 Vigade tuvastamine

6.1 Süntaksivead

Kõik lähtekoodi konstruktsioonid, mis ei vasta selle keele grammatikale on süntaksi vead ning need on tuvastatavad lähtekoodi parsimisel. Levinud süntaksivigadeks on näiteks puuduvad sulud ja semikoolonid. Neid vigu on küll lihtne parandada, kuid kui puudub kompilaator, mis neid kontrolliks, võivad need kergesti avastamata jääda. Seetõttu on hea, kui JavaScripti analüsaator ka need vead tuvastab.

ANTLR-i poolt genereeritud parser raporteerib süntaksivigu erinditena. Need erindid pärivad kõik klassi „CommonErrorNode". Kuna „CommonErrorNode" on „CommonTree" alamklass, siis saab need erindid lisada tippudena ka abstraktsesse süntaksi puusse [**43**]. JavaScripti analüsaatoris lisataksegi süntaksivea tekkimisel see AST-i ning elemendid, mida ei õnnestunud vea tõttu tuvastada, jäetakse lihtsalt vahele. AST-i puu transleerimisel XML-i lisatakse need vead DOM puu tippudeks tüübiga „ERROR" ning selle analüüsimise käigus lisatakse need tipud ka analüüsi tulemuste XML faili.

Süntaksivigade raporteerimist illustreerib järgmine näide. Olgu JavaScripti analüsaatori sisendiks programm, milles defineeritakse muutuja „i" ning tahetakse see väärtustada suurendamisoperaatoriga järgmiselt:

```
var i=++;
```

Kuna JavaScripti süntaks ei luba operaatorit muutujale väärtustada, siis annab JavaScripti parser süntaksivea kujul:

```
<unexpected: [@5,8:8=';',<102>,1:8], resync=;>
```

Veateade ütleb, et esimesel rea kaheksas sümbol on „**;**", aga grammatika järgi peaks olema midagi muud, näiteks sobiks, kui suurendamisoperaatorile järgneks muutuja, mida tahetakse suurendada. See süntaksiviga lisatakse lähtekoodi transleerimise tulemusena saadud XML dokumenti, nii nagu kujutatud näidisel 3.

```
<PROGRAM>
  <VAR>
    <OPERATOR value="=">
      <IDENTIFIER value="i" />
      <OPERATOR value="++">
        <ERROR
value="&lt;unexpected: [@5,8:8=';',&lt;102&gt;,1:8], resync=;&gt;" />
      </OPERATOR>
    </OPERATOR>
  </VAR>
</PROGRAM>
```

Näidis 3. Süntaksi viga XML dokumendis

6.2 Meetrikad

Tarkvara kvaliteedi meetrika (*software quality metric*) on IEEE 1061 standardi järgi defineeritud kui funktsioon, mille sisendiks on tarkvara lähtekood ning väljundiks on üks numbriline väärtus, mida saab tõlgendada, kui omadust, mis on mõjutatud selle tarkvara kvaliteedist [**44**]. Selle definitsiooni järgi annavad tarkvara meetrikad koodi kvaliteedile küll kvantitatiivse hinnangu, kuid jätavad selle arendaja tõlgendada.

JavaScripti analüsaator sisaldab struktuurseid tarkvara meetrikaid: tsüklomaatiline keerukus (*cyclomatic complexity*), hargnemiste sügavus (*depth of conditional nesting*) ja tsüklite sügavus (*depth of looping*) [**45**]. Neid meetrikaid rakendatakse igale programmi funktsioonile eraldi ning saadud tulemused aitavad hinnata, kas kood vajab ümberstruktureerimist või mitte. JavaScripti analüsaator annab ka hinnangu nende meetrikate väärtuste tõlgendamiseks viie riskitasemega, milles 0 tähistab minimaalset ja 4 maksimaalset riski. Riskitasemete määramisel on tuginetud Aivosto staatilise analüsaatori [**46**] hinnangutel. Järgnevalt on neid meetrikaid lähemalt käsitletud.

6.3 Tsüklomaatiline keerukus

Tsüklomaatiline keerukus on Thomas J. McCabe poolt loodud tarkvara meetrika, mis annab indikatsiooni koodi hargnevuse kohta. Tsüklomaatilist keerukust rakendatakse enamasti programmi meetoditele ning selle väärtus näitab ka soovitusliku test-stsenaariumite arvu selle programmi testimiseks [**47**]. Tsüklomaatiline keerukus on leitav järgmise avaldisega:

```
CC = hargnemiste arv + 1
```

Tsüklomaatilise keerukuse arvutamisel on erinevaid variatsioone, mis erinevad peamiselt selle poolest, kuidas leitakse hargnemise arv valemis. Nendeks variatsioonideks on täiendatud (*modified*), range (*strict*) ja põhiline (*essential*) tsüklomaatiline keerukus [**48**].

JavaScripti analüsaatoris on olemas meetrikad nii hariliku kui ka range tsüklomaatilise keerukuse arvutamiseks. Tavalise tsüklomaatilise keerukuse puhul käsitletakse hargnemistena tingimusi: „`if`“, „`else if`“, „`case`“, „`for`“, „`do-while`“, „`while`“, „`catch`“ ja valikuoperaator „`:?`“. Range tsüklomaatilise keerukuse puhul lisanduvad neile loogilised operaatorid: „`&&`“ ning „`||`“.

Tabelis 3 on kujutatud tsüklomaatilise keerukuse tõlgendamist JavaScripti analüsaatoris, mis baseerub Aivosto staatilise koodianalüsaatori hinnangutel [**46**].

Tabel 3. Tsüklomaatilise keerukuse tulemuste tõlgendamine

Tsüklomaatilise keerukuse väärtus	**Funktsiooni tüüp**	**JavaScripti analüsaatori riski tase**
1 - 4	lihtne funktsioon	0
5 - 10	stabiilne funktsioon	1
11 - 20	keerukam funktsioon	2
21 - 50	keeruline funktsioon	3
>50	äärmiselt keeruline funktsioon	4

6.4 Hargnemiste sügavus

Hargnemiste sügavus on tarkvara meetrika, mis on seotud tsüklomaatilise keerukusega. Kui tsüklomaatiline keerukus, näitab seda, kui palju hargnemisi koodis on, siis hargnemiste sügavus näitab seda, kui tugevasti on need hargnemised trepitud. Maksimaalne soovituslik hargnemiste sügavus funktsioonis on Aivosto staatilise analüsaatori hinnangul 5 [**46**]. JavaScripti analüsaator hindab funktsiooni riskitasemega 4, kui selle hargnemiste sügavus on enam kui 5 ning riskitasemega 2, kui selle hargnemiste sügavuseks on täpselt 5.

6.5 Tsüklite sügavus

Tsüklite sügavus on tarkvara meetrika, mis sarnaneb hargnemiste sügavusele, kuid sügavuse leidmisel vaadeldakse ainult tsükleid: „`for`“, „`do-while`“ ja „`while`“. Maksi-

maalne soovituslik tsüklite sügavus funktsioonis on Aivosto staatilise analüsaatori hinnangul 2 [**46**]. JavaScripti analüsaator hindab funktsiooni riskitasemega 4, kui selle tsüklite sügavus on enam kui 2 ning riskitasemega 2, kui selle tsüklite sügavuseks on täpselt 2.

6.6 Halvad praktikad

W3C konsortsiumis on kirjeldatud halva praktikana optimeerimata tsükleid [**49**], mille esinemisi kontrollib ka JavaScripti analüsaator. Kuivõrd erinevaid vigu ja halbasid praktikaid on teisigi, siis on selle kontrolli eesmärk näidata JavaScripti analüsaatori võimekust selliseid vigu tuvastada.

JavaScriptis loetakse halvaks praktikaks objekti atribuudi lugemist tsükli sees. Kuna JavaScript on dünaamiliselt tüübitud keel, on tegemist aeglase operatsiooniga. Kõige enam eksitakse selle praktika vastu järjendite läbimisel, mida tehakse enamasti nii, nagu näidisel 4 kujutatud.

```
for (var i=0; i<järjend.length; i++) {
  ...
}
```

Näidis 4. Järjendi pikkuse lugemine igal iteratsioonil

Probleemi lahendamiseks piisab sellest, kui defineerida järjendi pikkus väljaspool tsüklit, nagu on kujutatud näidisel 5.

```
var pikkus = järjend.length;
for (var i=0; i<pikkus; i++) {
  ...
}
```

Näidis 5. Järjendi pikkus defineerimine väljaspool tsüklit

Selle vea leidmiseks kontrollib JavaScripti analüsaator, kas tsükli jätkamistingimuses on pöördutud objekti atribuutide poole. See kontroll teostatakse funktsioonide põhiselt ning probleemi ilmnemisel lisatakse tulemuste XML dokumenti element kujul:

```
<badstyle name="unoptimized_loop" loop_type="FOR" property_name="length"
description="Unoptimized loop FOR. Create additional variable for length
value in pre-loop statement."/>
```

6.7 Näide JavaScripti analüsaatori kasutamisest

Järgnevalt on toodud näide JavaScripti analüsaatori tööst. Näide on tehtud JavaScripti analüsaatori silumisversiooni peal ning seetõttu salvestatakse analüüsi vahetulemused programmi töö kausta. Lõppversioonis vahetulemusi eraldi ei salvestata.

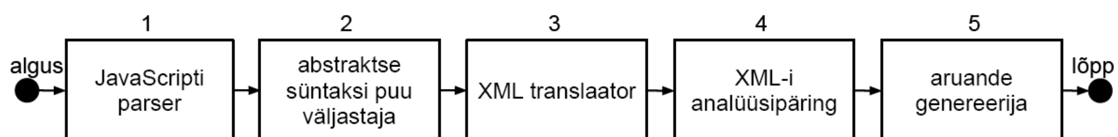
Analüüsitavaks programmiks on valitud mulli meetodil põhinev sorteerimisalgoritm [**50**], mille lähtekoodi on kujutatud näidisel 6.

```
var a = [5, 1, 3, 2, 4];
bubbleSort(a);

function bubbleSort(list) {
  var swapped;
  do {
    swapped = false;
    for (var i=0; i<list.length-1; i++) {
      if (list[i] > list[i+1]) {
        var temp = list[i];
        list[i] = list[i+1];
        list[i+1] = temp;
        swapped = true;
      }
    }
  } while (swapped);
}
```

Näidis 6. Järjendi sorteerimine mulli meetodiga

Teenuse käivitamisel saadetakse kliendile pistikprogramm ja alustatakse joonisel 4 kujutatud töövoogu.



Joonis 4. JavaScripti analüsaatori töövoog

Esimeses komponendis käivitatakse JavaScripti parser, mis teisendab lähtekoodi abstraktseks süntaksi puuks.

Teises komponendis käivitatakse abstraktse süntaksi puu väljastaja, mis kirjutab parseri poolt genereeritud puu konsooliaknasse. Lõppversioonis abstraktset süntaksi puud ei väljastata, sest analüsaatori kasutajal pole seda otseselt tarvis.

Kolmandas komponendis käivitatakse XML-i translaator, mis transleerib abstraktse süntaksi puu DOM puuks. Selle tulemusena saadud XML dokument salvestatakse töövoos defineeritud nimega faili, milleks on vaikimisi „outputXML.xml". Lisas 2 on toodud sorteerimisalgoritmi abstraktsele süntaksi puule vastav XML dokument.

Neljandas komponendis käivitatakse XML-i analüüsipäring, mis teeb eelnevalt genereeritud XML dokumendi peal XSLT teisendused. Selle väljundiks on analüüsi tulemustega XML dokument, mis kannab vaikimisi nime „results.xml". Sorteerimisalgoritmile vastav resultaat XML fail on toodud näidisel 7.

```
<?xml version="1.0" encoding="utf-8"?>
<results>
  <elementCount>
    <count name="global_functions">1</count>
    <count name="global_variables">1</count>
    <count name="total_variables">4</count>
  </elementCount>
  <functions>
    <function name="bubbleSort">
      <count name="variables">3</count>
      <metric name="cyclomatic_complexity" result="4" risk_level="0"/>
      <metric name="strict_cyclomatic_complexity" result="4"
risk_level="0"/>
      <metric name="depth_of_conditional_nesting" result="3"
risk_level="0"/>
      <metric name="depth_of_looping" result="2" risk_level="2"/>
      <badstyle name="unoptimized_loop" loop_type="FOR"
property_name="length" description="Unoptimized loop FOR. Create
additional variable for length value in pre-loop statement."/>
    </function>
  </functions>
</results>
```

Näidis 7. JavaScripti analüsaatori analüüsi tulemuste fail

Tulemuste faili esimeses blokis „elementCount" on sisendfailis olevate globaalsete funktsioonide ja muutujate arv. Samuti on seal välja toodud kõigi muutujate arv kokku. Suuremaid projekte analüüsides on neid lihtsaid väärtuseid hea võrrelda. Tulemuste faili teises blokis „functions" on toodud kontrollide tulemused kõigi globaalsete funktsioonide jaoks. Antud näite puhul on ainult üks funktsioon „bubbleSort", mida on analüüsitud. Selle funktsiooni kohta on arvutatud lokaalsete muutujate arv ning eespool kirjeldatud koodimeetrikad. Samuti leidis analüüsipäring ühe optimeerimata `for` tsükli.

Viimases komponendis koostab JavaScripti analüsaator tulemuste XML dokumendist kompaktse HTML kujul aruande. Sorteerimisalgoritmile vastav aruanne on kujutatud joonisel 5.

JavaScript analyzer report

Overview

global functions: 1

global variables: 1

total variables: 4

Functions

function name	variables	cyclomatic complexity	strict cyclomatic complexity	depth of conditional nesting	depth of looping
bubbleSort	3	4	4	3	2
	Bad style:	Unoptimized loop FOR. Create additional variable for length value in pre-loop statement.			

Joonis 5. JavaScripti analüsaatori genereeritud aruanne

7 Järeldused

JavaScripti analüsaatori loomiseks valitud platvorm on staatilise analüüsi teostamiseks sobiv ja efektiivne. Valminud analüsaatorit saaks kohandada ka teistele keeltele. Selleks tuleks asendada töövoos JavaScripti parser mõne teise keele parseriga, mis koostaks samasuguse struktuuriga abstraktse süntaksi puu.

Parserigeneraator ANTLR genereerib piisavalt kiireid parsereid, et isegi mahukamate koodibaaside parsimiseks kuluvad vaid mõned sekundid. Samuti on piisavalt kiire ka XSLT teisendustel põhinev analüüsimeetod. ANTLR-i kolmanda versiooni puuduseks on see, et elementide asukohtasid lähtekoodis ei salvestata AST-i. AST tippude liidestes on küll olemas vastavad meetodid rea ja veeru küsimiseks, kuid vaikimisi tagastatakse nende väärtuseks null. Seda probleemi saab lahendada, kui defineerida uus AST tipu klass, mis pärib klassi „CommonTree“ ja lisaks salvestab endale rea ja veeru numbrid [**51**].

Staatilise koodianalüsaatori puhul on oluline, et selles teostatavad kontrollid oleksid värsked ning analüüsi tulemusi hinnataks õigesti. Nagu enamik mustrite sobitamisel põhinevaid staatilisi koodianalüsaatoreid, pole ka JavaScripti analüsaator „mõistlik“ ega „täielik“. Kuna JavaScripti analüsaator kontrollib ainult globaalseid funktsioone, siis ei leia ta näiteks optimeerimata tsükleid juhul, kui need asuvad objektide funktsioonides.

Et kasutajatel oleks JavaScripti analüsaatorist rohkem kasu, on tarvis selle kontrolle jooksvalt täiendada. Selle hõlbustamiseks saab kasutada teenusepõhist lähenemist, mis võimaldab neid kontrolle mugavalt muuta ja kasutajate vahel jagada. Lisaks saab serveri poolel koguda olulist analüütikat, mida kontrollide täiendamiseks kasutada.

Kokkuvõte

Käesoleva töö raames anti ülevaade JavaScripti staatilistest koodianalüsaatoritest. Samuti käsitleti JavaScripti staatilise analüsaatori loomist teenusepõhise lahendusena, milles serveri juhitud analüüs viiakse läbi kliendi poolel. Sellise lähenemise eeliseks on see, et teenuse poolel on muudatuste tegemine lihtsam kui eraldiseisvas rakenduses ning need muudatused jõuavad aegsasti ka lõppkasutajateni. Seejuures saab keskses serveris koguda olulist analüütikat, mida analüsaatori täiendamiseks kasutada.

Töö raames valmis prototüüprakendus teenusena toimivast JavaScripti analüsaatorist. Selle loomiseks kasutati parserigeneraatorit ANTLR, mille abil loodi translaator JavaScripti lähtekoodikoodi transleerimiseks XML dokumendiks. XML dokumendi töötlemiseks on olemas standardvahendid XPath ja XSLT, mida on JavaScripti analüsaatoris kasutatud lähtekoodi kvaliteedi hindamiseks.

JavaScripti analüsaator kontrollib lähtekoodi semantilist korrektsust, arvutab sellele tuntud koodimeetrikate väärtusi ning oskab tuvastada halbu praktikaid. Saadud tulemused hindavad eelkõige lähtekoodi struktuuri keerukust ning annavad nõu selle parandamiseks. Need tulemused esitatakse kasutajale kompaktse aruandena, mille eesmärk on anda tervikpilt lähtekoodi kvaliteedi kohta.

Tulevikuarendustena tuleks JavaScripti analüsaatorit testida avatud lähtekoodiga projektide peal ning saadud tulemuste põhjal täiendada selles teostatavate kontrollide komplekti. Serveris talletatud tulemuste analüüsimiseks võiks kasutada masinõppe meetodeid, mille abil luua automaatselt kohanduv mudel analüüsitulemuste tõlgendamiseks. See mudel võiks asuda serveri poolel, tõstes seeläbi teenuse osatähtsust analüüsi protsessis. Töövool põhinev analüsaatori struktuur võimaldab selle komponente ka väiksemate osadena välja vahetada ja seeläbi on võimalik JavaScripti analüsaatorit kohandada ka teiste keelte analüüsimiseks.

Creating JavaScript static code analyser as a service

Bachelor's Thesis (6 ECTS)

Jaanus Jaggo

Summary

Static code analysis is a widespread analysis method used in software industry. It helps to estimate software quality and also to conduct code reviews in earlier stages of the software project. The majority of contemporary analysers are made as independent applications which contain fixed sets of patterns used to find bugs.

This thesis represents a service-oriented approach of developing a static code analyser using private source code principle which means that server controlled analysis is conducted in the client side. To demonstrate the sensibility and feasibility of this approach a prototype tool for analysing JavaScript source code is made.

The JavaScript analyser is made as a part of Siim Karus's XMLStyleHelper to extend it with the JavaScript source code analysis capability. Parser generator ANTLR is used to translate JavaScript source code to XML document. This document contains the structure of abstract syntax tree, which is analysed using standard languages, namely XPath and XSLT.

JavaScript analyser checks semantic validity of the source code, calculates software quality metrics, and identifies bad programming practices. The JavaScript analyser outputs an HTML document with analysis results to the user when it is finished with analysis.

The JavaScript analyser helps to estimate the complexity of code structure. It also gives an overview about functions in the software code. The service-oriented approach employed by the JavaScript analyser has several advantages compared to standalone analysis tools and thanks to private source code principle it is also more acceptable for potential users.

Viited

[1] Ashfaque Ahmed, *Software testing as a service*. New York: Taylor & Francis Group, 2009.

[2] M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal*, 182-211, 1976.

[3] Michael Ernst, "Static and dynamic analysis: synergy and duality," - ICSE Workshop on Dynamic Analysis (WODA), 35-35, 2003.

[4] Ciera Jaspan, I-Chin Chen ja Anoop Sharma, "Understanding the value of program analysis tools," - *OOPSLA '07 Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, New York, 963-970, 2007.

[5] Thomas Ball, "The SLAM project: debugging system software via static analysis," - *POPL '02 Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, 1-3, 2002.

[6] McGraw Gary, "Static analysis for security," *Security & Privacy, IEEE*, vol 2, nr 6, 76-79, 2004.

[7] Jernej Novak, Andrej Krajnc ja Rok Zontar, "Taxonomy of static code analysis tools," - *MIPRO, 2010 Proceedings of the 33rd International Convention*, Maribor, 418-422, 2010.

[8] Mikko Vestola, "Evaluating and enhancing FindBugs to detect bugs from mature software: Case study in Valuatum," Aalto University, Espoo, magistritöö 2012.

[9] Gregor Richards, Sylvain Lebresne, Brian Burg ja Jan Vitek, "An analysis of the dynamic behavior of JavaScript programs," - *2010 ACM SIGPLAN conference on Programming language design and implementation*, New York, 1-12, 2010.

[10] Brian Hackett ja Shu-yu Guo, "Fast and Precise Hybrid Type Inference for JavaScript," - *PLDI '12 Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, New York, 239-250, 2012.

[11] Google Code, V8 engine. [Online]. http://code.google.com/p/v8 (kasutatud 29 aprill 2013)

[12] Douglas Crockford, JSLint. [Online]. http://www.jslint.com/ (kasutatud 1. mai 2013)

[13] JSMeter. [Online]. https://code.google.com/p/jsmeter/ (kasutatud 1. mai 2013)

[14] Paruj Ratanaworabhan, Benjamin Livshits ja Benjamin G Zorn, "JSMeter: comparing the behavior of JavaScript benchmarks with real web applications," - *WebApps'10 Proceedings of the 2010 USENIX conference on Web application development*, California, California, 3-3, 2010.

[15] Sourceforge, PMD project page. [Online]. http://pmd.sourceforge.net/ (kasutatud 2. mai 2013)

[16] Ciera Nicole Christopher, "Evaluating Static Analysis Frameworks," Carnegie Mellon University, Pittsburgh, kursusetöö 2006.

[17] Jaroslav Porubän, Miroslav Sabo, Ján Kollár ja Marjan Mernik, "Abstract Syntax Driven Language Development: Defining language semantics through aspects," - *International Workshop on Formalization of Modeling Languages*, New York, 2010.

[18] Keeleveeb, IT terministandardi projekti (1998-2001) sõnastik [Online]. http://www.keeleveeb.ee/dict/speciality/itstandard/dict.cgi?lang=en&word=parser (kasutatud 10. märts 2013)

[19] Jaak Henno, *Formaalsed keeled, grammatikad ja translaatorid*. Tallinn: Tallinna Tehnikaülikooli kirjastus, 2006.

[20] Ian Kaplan. Why Use ANTLR? [Online]. http://www.bearcave.com/software/antlr/antlr_expr.html (kasutatud 20. aprill 2013)

[21] Terence Parr, ANTLR 3 [Online]. http://www.antlr3.org/about.html (kasutatud 4. märts 2013)

[22] Lars Marius Garshol. „BNF and EBNF: What are they and how do they work?“ [Online]. http://www.garshol.priv.no/download/text/bnf.html (kasutatud 7. aprill 2013)

[23] Terence Parr ja Kathleen Fisher, "LL(*): The Foundation of the ANTLR Parser Generator," - *PLDI '11 Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, New York, 425-436, 2011.

[24] Steven V. Earhart, *UNIX programmer's manual*. New York: Holt McDougal, 1986.

[25] Ain Isotamm, *Translaatorite tegemise süsteem*. Tartu: Tartu Ülikooli kirjastus, 2012.

[26] Noam Chomsky, . "Three models for the description of language" - *Information Theory, IRE Transactions on*, vol 2, nr 3, 1956.

[27] ISO Store. ISO/IEC 14977:1996 [Online]. http://www.iso.org/iso/catalogue_detail.htm?csnumber=26153 (kasutatud 7. aprill 2013)

[28] Mozilla Corporation. New in JavaScript 1.7 [Online]. https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7 (kasutatud 1. mai 2013)

[29] Ecma International. ECMAScript® Language Specification [Online]. http://www.ecma-international.org/ecma-262/5.1/ (kasutatud 4. märts 2013)

[30] Jaak Henno, Kirjavahetus 27 aprill 2013.

[31] Chris Lambrou. License clarification - ECMAScript grammar, JavaScript.g [Online]. http://www.antlr3.org/pipermail/antlr-interest/2008-April/027771.html (kasutatud 4. märts 2013)

[32] Jeni Tennison, *Beginning XSLT 2.0 From Novice to Professional*. California: Apress, 2005.

[33] Michael Kay, *XSLT 2.0 and XPath 2.0 Programmer's Reference 4th Edition*. Indianapolis: Wiley Publishing, Inc., 2008.

[34] Stephan Kepser, "A Proof of the Turing-completeness of XSLT and XQuery," Eberhard Karls Universitat Tubingen ,Tubingen, tehniline aruanne, 2002.

[35] Terence Parr, *The Definitive ANTLR Reference : Building Domain-Specific Languages*. Dallas: Pragmatic Bookshelf, 2007.

[36] Sourceforge. Findbugs. [Online]. http://findbugs.sourceforge.net/ (kasutatud 2. mai 2013)

[37] Ivo Gomes, Pedro Morgado, Tiago Gomes ja Rodrigo Moreira. An overview on the Static Code Analysis approach in Software Development [Online]. http://paginas.fe.up.pt/~ei05021/TQSO%20-%20An%20overview%20on%20the%20Static%20Code%20Analysis%20approach%20in%20Software%20Development.pdf (kasutatud 21. veebruar 2013)

[38] Gustavo Alonso, Fabio Casati, Harumi Kuno ja Vijay Machiraju, *Web Services, Concepts, Architectures and Applications*. Berliin: Springer, 2004.

[39] Keith Bennett et al., "Service-based software: the future for flexible software," - *Software Engineering Conference, 2000. APSEC 2000. Proceedings. Seventh Asia-Pacific*, Los Alamitos, 214-221, 2000.

[40] Siim Karus, „XML development with plug-ins as a service“ - *Developing Tools as Plug-ins (TOPI), 2012 2nd Workshop on*, Tartu, 25-30, 2012.

[41] David Mann, *Workflow in the 2007 Microsoft Office System*. California: Apress, 2007.

[42] Marco Zapletal, Wil M. P. van der Aalst, Nick Russell, Philipp Liegl ja Hannes Werthner, "Pattern-based Analysis of Windows Workflow," Vienna University of Technology, Austria, tehniline aruanne, 2009.

[43] Terence Parr. (2013, mai) ANTLR 3 Reference Manual. [Online]. http://www.antlr3.org/share/1084743321127/ANTLR_Reference_Manual.pdf (kasutatud 25. märts 2013)

[44] Institute of Electrical and Electronics Engineers, Inc, „IEEE Standard for a Software Quality Metrics Methodology“, New York, 1998.

[45] Lu Wei, "Supervised Categorization of JavaScriptTM using Program," National University of Singapore, Singapore, projekti aruanne, 2005.

[46] Aivosto Oy. Complexity metrics. [Online]. http://www.aivosto.com/project/help/pm-complexity.html (kasutatud 8. mai 2013)

[47] Thomas J. McCabe , "A complexity measure," - *ICSE '76 Proceedings of the 2nd international conference on Software engineering*, Los Alamitos, 308-320, 1976.

[48] Istehad Chowdhury ja Mohammad Zulkernine, "Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities," - *Journal of Systems Architecture: the EUROMICRO Journal*, vol 57, nr 3, 294-313, 2011.

[49] World Wide Web Consortium. JavaScript best practices. [Online]. http://www.w3.org/wiki/JavaScript_best_practices (kasutatud 8. mai 2013)

[50] Wikipedia. Bubble sort. [Online]. http://en.wikipedia.org/wiki/Bubble_sort (kasutatud 8. mai 2013)

[51] Alex Miller. Recovering line and column numbers in your Antlr AST. [Online]. http://tech.puredanger.com/2007/02/01/recovering-line-and-column-numbers-in-your-antlr-ast/ (kasutatud 8. mai 2013)

Lisa 1. Mõisted

Järgnevalt defineeritud mõisted pärinevad Ain Isotamme raamatust „Translaatorite tegemise süsteem“ [**25**].

Kontekstivaba grammatika

Üks võimalus masinast sõltumatute keelte formaalseks kirjeldamiseks on kasutada *kontekstivaba grammatikat* (KVG).

Kontekstivaba grammatika (KVG) on järjestatud nelik (V_N , V_T ,P ,S), kus

- V_N on mitteterminaalne (mõistete) tähestik,
- V_T on terminaalne tähestik,
- P on produktsioonide hulk kujul $A \rightarrow x$, kus $A \in V_N$ ja $x \in V^*$ ja
- S on aksioom: fikseeritud täht tähestikus V_N, mida ei saa kasutada teiste mõistete defineerimiseks.

Terminaalset tähestikku V_T nimetatakse ka lekseemide tähestikuks ning selle tähestiku elementi $x \in V_T$ nimetatakse lekseemiks. Lekseemid on kõik sümbolid, millest koosneb programmi nähtav tekst. Lekseemide alla kuuluvad reserveeritud sõnad, eraldajad, kommentaarid, identifikaatorid, konstandid ja stringid.

Mitteterminaalne tähestik V_N on defineeritavate mõistete hulk.

Kontekstivaba grammatika aksioomist S saame produktsioonide rakendamise teel tuletada kõik antud grammatikaga kirjeldatavad sõnad ehk programmid.

V^* tähistab kõikide tähestiku $V_N \cap V_T$ baasil moodustatavate sõnade hulka.

Kanooniline derivatsioon

Kanoonilise derivatsiooni igal sammul asendatakse kõige parempoolsem mitteterminal produktsiooni paremas pooles olevate elementidega. Derivatsiooni abil saab aksioomist lähtudes genereerida iga selles keeles defineeritava sõna.

Derivatsiooni puu

Aksioomile S produktsioone rakendades saame moodustada derivatsiooni puu.

Kontekstivaba grammatika G derivatsiooni puu on järgmiste reeglite kohaselt märgendatud tippudega puu:

- üksainus tipp märgendiga S (s.o. aksioom) on derivatsiooni puu;
- kui D on derivatsiooni puu ja N tema tipp märgendiga $A \in V_N$ ning$A \to X_1, \dots, X_p \in P$ saame moodustada uue derivatsiooni puu D', asendades tipu N märgendi uuega $A \to X_1, \dots, X_p \in P$ ja lisades tipule N p alluvat, millede märgendeiks paneme (vasakult paremale) $X_1, \dots, X_p$.

Grammatika ühesus

Kontekstivaba grammatikat nimetatakse üheseks, kui iga selles keeles defineeritud sõna saamiseks leidub parajasti üks kanooniline derivatsioon.

Lisa 2. Sorteerimisprogrammi AST

Järgnevalt on toodud XML kujul abstraktne süntaksi puu, mis on genereeritud käesoleva töö näidisel 6 kujutatud lähtekoodi põhjal.


```
<?xml version="1.0" encoding="utf-8"?>
<PROGRAM>
  <VAR>
    <OPERATOR value="=">
      <IDENTIFIER value="a" />
      <ARRAY>
        <NUMERIC value="5" />
        <NUMERIC value="1" />
        <NUMERIC value="3" />
        <NUMERIC value="2" />
        <NUMERIC value="4" />
      </ARRAY>
    </OPERATOR>
  </VAR>
  <CALL>
    <IDENTIFIER value="bubbleSort" />
    <ARGUMENT>
      <IDENTIFIER value="a" />
    </ARGUMENT>
  </CALL>
  <FUNCTION>
    <IDENTIFIER value="bubbleSort" />
    <PARAMETER>
      <IDENTIFIER value="list" />
    </PARAMETER>
    <FUNCTIONBODY>
      <VAR>
        <IDENTIFIER value="swapped" />
      </VAR>
      <DOWHILE>
        <IDENTIFIER value="swapped" />
        <BLOCK>
          <OPERATOR value="=">
            <IDENTIFIER value="swapped" />
            <BOOLEAN value="false" />
          </OPERATOR>
          <FOR>
            <VAR>
              <OPERATOR value="=">
                <IDENTIFIER value="i" />
                <NUMERIC value="0" />
              </OPERATOR>
            </VAR>
            <CONDITION>
              <OPERATOR value="&lt;">
                <IDENTIFIER value="i" />
```

```
    <OPERATOR value="-">
      <IDENTIFIER value="list" />
      <PROPERTY>
        <IDENTIFIER value="length" />
      </PROPERTY>
      <NUMERIC value="1" />
    </OPERATOR>
  </OPERATOR>
</CONDITION>
<ITERATE>
  <IDENTIFIER value="i">
    <OPERATOR value="++" />
  </IDENTIFIER>
</ITERATE>
<BLOCK>
  <IF>
    <OPERATOR value="&gt;">
      <IDENTIFIER value="list" />
      <INDEX>
        <IDENTIFIER value="i" />
      </INDEX>
      <IDENTIFIER value="list" />
      <INDEX>
        <OPERATOR value="+">
          <IDENTIFIER value="i" />
          <NUMERIC value="1" />
        </OPERATOR>
      </INDEX>
    </OPERATOR>
    <BLOCK>
      <VAR>
        <OPERATOR value="=">
          <IDENTIFIER value="temp" />
          <IDENTIFIER value="list" />
          <INDEX>
            <IDENTIFIER value="i" />
          </INDEX>
        </OPERATOR>
      </VAR>
      <OPERATOR value="=">
        <IDENTIFIER value="list" />
        <INDEX>
          <IDENTIFIER value="i" />
        </INDEX>
        <IDENTIFIER value="list" />
        <INDEX>
          <OPERATOR value="+">
            <IDENTIFIER value="i" />
            <NUMERIC value="1" />
          </OPERATOR>
        </INDEX>
      </OPERATOR>
      <OPERATOR value="=">
        <IDENTIFIER value="list" />
```

```
                    <INDEX>
                      <OPERATOR value="+">
                        <IDENTIFIER value="i" />
                        <NUMERIC value="1" />
                      </OPERATOR>
                    </INDEX>
                    <IDENTIFIER value="temp" />
                  </OPERATOR>
                  <OPERATOR value="=">
                    <IDENTIFIER value="swapped" />
                    <BOOLEAN value="true" />
                  </OPERATOR>
                </BLOCK>
              </IF>
            </BLOCK>
          </FOR>
        </BLOCK>
      </DOWHILE>
    </FUNCTIONBODY>
  </FUNCTION>
</PROGRAM>
```

Lisa 3. DVD bakalaureusetöö juurde kuuluvate failidega

Käesoleva bakalaureusetöö juurde kuulub DVD, mis sisaldab järgmiseid faile:

1. JavaScripti analüsaator
2. JavaScripti analüsaatori lähtekood
3. JavaScripti grammatika
4. Veakontrollidele vastavad XSLT mallid

Lihtlitsents lõputöö reprodutseerimiseks ja lõputöö üldsusele kättesaadavaks tegemiseks

Mina, Jaanus Jaggo (sünnikuupäev: 02.03.1991)

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) enda loodud teose „JavaScripti staatilise koodianalüsaatori loomine teenusena“, mille juhendajad on Siim Karus ja Sven Laur,

1.1. reprodutseerimiseks säilitamise ja üldsusele kättesaadavaks tegemise eesmärgil, sealhulgas digitaalarhiivi DSpace-is lisamise eesmärgil kuni autoriõiguse kehtivuse tähtaja lõppemiseni;

1.2. üldsusele kättesaadavaks tegemiseks Tartu Ülikooli veebikeskkonna kaudu, sealhulgas digitaalarhiivi DSpace´i kaudu kuni autoriõiguse kehtivuse tähtaja lõppemiseni.

2. olen teadlik, et punktis 1 nimetatud õigused jäävad alles ka autorile.

3. kinnitan, et lihtlitsentsi andmisega ei rikuta teiste isikute intellektuaalomandi ega isikuandmete kaitse seadusest tulenevaid õigusi.

Tartus, 13.05.2013