TARTU ÜLIKOOL
Arvutiteaduse instituut
Informaatika õppekava

Karmo Saviauk

Automaattestide kirjapanemise keele TSL edasiarendamine

Bakalaureusetöö (9 EAP)

Juhendaja Reimo Palm, PhD

Tartu 2025

Automaattestide kirjapanemise keele TSL edasiarendamine

Lühikokkuvõte

Automaatkontrollid on programmeerimise õpetamise oluline osa, kuna need annavad tudengitele kiiret tagasisidet ja vähendavad õppejõudude töökoormust. Käesoleva töö sisuks on Tartu Ülikoolis kasutatava domeenispetsiifilise automaattestide kirjapanemise keele TSL (Test Specific Language) edasiarendamine. Töö käigus analüüsiti Tartu Ülikooli kursuse „Programmeerimine“ kõige sagedamini lahendatavate ülesannete klasse ning nende lahenduste kontrollimise põhimõtteid ja strateegiaid. TSL-keele täiustamiseks arendati välja uusi testitüüpe, täiustati olemasolevaid ning muudeti tagasiside esitus õppijale informatiivsemaks. Kõik muudatused lisati TSL-keelde, laiendades selle kasutusvõimalusi ja parandades praktilist rakendatavust. Töö tulemusel paranes süsteemi võime tuvastada levinumaid vigu, hinnata keerukamaid lahendusi, pakkuda õppijale selgemat ja sisukamat tagasisidet ning vähendada õppejõu töökoormust.

Võtmesõnad: programmeerimine, programmeerimise kursus, automaatkontroll, Test Specific Language

CERCS: P175 Informaatika, süsteemiteooria, S281 Arvuti õpiprogrammide kasutamise metoodika ja pedagoogika

Extending the TSL Language for Writing Automated Tests

Abstract

Automated assessment plays a key role in programming education by providing immediate feedback to students and reducing the workload of instructors. This thesis focuses on the development of TSL (Test Specific Language), a domain-specific language used at the University of Tartu for writing automated tests. The work includes an analysis of the most common types of programming assignments in the "Computer Programming" course and the corresponding assessment strategies. To improve the TSL language, new test types were implemented, existing ones were enhanced, and the quality of feedback provided to students was improved. All changes were integrated into the TSL language to expand its capabilities and improve its practical applicability. As a result of this work, the system became more capable of detecting common errors, evaluating more complex solutions, providing clearer and more meaningful feedback to students, and reducing the workload of instructors.

Keywords: programming, programming course, automated tests, Test Specific Language

CERCS: P175 Informatics, systems theory, S281 Computer-assisted education

Sisukord

Sissejuhatus 5
1 Programmide automaatkontrollid 6
1.1 Automaatkontrollide olulisus 6
1.2 Programmeerimisülesannete klassifikatsioon 7
1.2.1 Tüübi järgi 7
1.2.2 Vormi järgi 9
1.3 Automaatkontrollimise põhimõtted ja strateegiad 10
2 TSL 12
2.1 TSLi automaatkontrollide töövoog 12
2.2 TSLi testide struktuur 15
2.3 TSLi edasiarendamise vajadus 16
3 Metoodika 18
3.1 Töökeskkonnad 18
3.2 Arendusprotsess 18
4 Tulemused 20
4.1 Olemasolevate testitüüpide parandused 20
4.1.1 Väljundi kontroll 20
4.1.2 Funktsiooni staatiline test 21
4.1.3 Klassi dünaamiline test 21
4.1.4 Funktsiooni dünaamiline test 21
4.1.5 Süsteemi poolt tagastatud tulemused 23
4.2 Uued testitüübid 24
4.2.1 Fraasi olemasolu test 24
4.2.2 Klassi päriluse testid 24
4.2.3 Klassi isendite ja meetodite staatilised testid 25
4.2.4 Põhiprogrammi testid 26
4.3 Uus süsteem 26
5 Testimine 28
6 Edasiarenduse võimalused 30
7 Kokkuvõte 31
8 Viidatud kirjandus 32
Lisad 35
Lisa 1 TSLis kasutatavad testitüübid 35
Lisa 1.1 Senised testitüübid 35
Lisa 1.2 Töö käigus loodud uued testitüübid 36
Lisa 2 Valminud tarkvara 38
Litsents 39

Sissejuhatus

Tänapäeva programmeerimisõpe on tihedalt seotud automaatsete hindamissüsteemidega, mis võimaldavad anda efektiivset ja objektiivset tagasisidet õppijatele. Ülikoolides, kus õppetöös kasutatakse järjest enam programmeerimisülesannete automaatkontrolle, on oluline nende süsteemide täpsus, tagasiside kiirus ja asjakohasus. Automaatsete testimissüsteemide eesmärk ei ole mitte ainult hindamisprotsessi kiirendamine, vaid ka tudengite programmeerimisoskuste arendamine kiire tagasiside ja vigade analüüsi kaudu.

Käesolev bakalaureusetöö keskendub Tartu Ülikooli kursusel „Programmeerimine" kasutatava automaattestide kirjapanemise keele TSL edasiarendamisele. Kuigi TSL on juba kasutusel, on selle funktsionaalsuses ilmnenud mitmeid kitsaskohti, mida saab parandada, et hindamine oleks täpsem ja paindlikum. Töö käigus ei keskendutud üksnes olemasolevate kitsaskohtade parandamisele, vaid täiendati TSLi ka uute vahenditega, et suurendada keele praktilist kasutusulatust ja kohandatavust eri ülesandetüüpide hindamisel.

Töö eesmärk on TSL-keele täiustamine, võimaldamaks programmeerimisülesannete täpsemat hindamist. Selleks analüüsitakse esmalt kursusel „Programmeerimine" kõige sagedamini lahendatavate ülesannete klasse ning nende kontrollimise põhimõtteid ja strateegiaid. Seejärel tutvustatakse TSL-keele ülesehitust ja põhjendatakse selle edasiarendamise vajadust. Kirjeldatakse uusi loodud testitüüpe, olemasolevate testitüüpide täiendusi ja õppijale antava tagasiside parandusi. Lõpetuseks hinnatakse tehtud muudatuste mõju keele funktsionaalsusele ja paindlikkusele ning pakutakse välja edasised arendusvõimalused. Töö tulemusel paranes süsteemi võime tuvastada levinumaid vigu, hinnata keerukamaid lahendusi, pakkuda õppijale selgemat ja sisukamat tagasisidet ning vähendada õppejõu töökoormust.

1 Programmide automaatkontrollid

Selles peatükis antakse ülevaade automaatkontrollidest, erinevatest programmeerimisülesannete tüüpidest ning nende automaatseks kontrollimiseks kasutatavatest strateegiatest.

1.1 Automaatkontrollide olulisus

Araujo [1] defineerib automaatset hindamist kui süsteemi, mis võimaldab õpetajatel või õppeasutustel õpilaste töid tõhusalt ja kiirelt hinnata. Automaatse hindamise protsess hõlmab ülesannete automaatset hindamist arvutipõhiste algoritmide abil, mis annavad õpilastele kiiret ja objektiivset tagasisidet.

Programmide automaatsel kontrollimisel on võrreldes käsitsi läbivaatamisega mitmeid eeliseid. Programmide käsitsi kontrollimine on ajakulukas. Joosep Albre [2] on välja toonud, et Tartu Ülikooli kursusel „Programmeerimine“ [3] tuleks õppejõududel kursuse vältel kontrollida ligikaudu 45000 ülesannet. Seda üleliigset tööd aitab vältida programmide automaatne kontrollimine. Samuti ei pea õpilased ootama õppejõu tagasisidet, vaid saavad oma tulemused teada sekunditega [1].

Cheanga [4] on välja toonud, et inimesel on keeruline hinnata programmi korrektsust, sest see nõuab tihti koodi rida-realt läbi vaatamist või programmi korduvat täitmist erinevate sisenditega. Automaatne hindamine võimaldab täpselt ja kiiresti hinnata programmi korrektsust erinevate testandmetega.

Käsitsi kontrollimisel võivad eri hindajad anda samale programmile erinevad hinded ning isegi sama hindaja võib olenevalt meeleolust või tähelepanust hinnata programme erinevalt [4]. Automaattestidel seda probleemi ei esine, sest neil on ühtne ja objektiivne hindamisloogika.

Ülikoolides on olulisel kohal ka plagiaadituvastus. Programmi käsitsi testides võib plagiaat jääda märkamata isegi juhul, kui hindamisprotsessis osaleb mitu inimest. Plagiaadi tuvastamiseks saab kasutada tarkvara, mis võrdleb tudengite lahendusi ja püüab tuvastada võimalikku plagiaati [5]. Sama põhimõttega lahendust kasutab ka Tartu Ülikooli kursus „Programmeerimine“ keskkonnas Lahendus [6], kus saab ühe nupuvajutusega leida kõige sarnasemate tööde paarid.

Automaatkontrollidest on kasu ka tudengitele. Näiteks võimaldab automaatne hindamine tudengitel oma lahendusi korduvalt esitada, saades seeläbi ühtlasi ka kiiret tagasisidet [7]. See omakorda aitab neil õppida oma vigadest ja kinnistada teadmisi. Kuna tudeng saab

ebaõnnestunud testidest kohese ülevaate ja peab leidma ise vea põhjuse, toetab see protsess oskust iseseisvalt vigu tuvastada ja parandada.

Automaatkontroll soodustab ülesannete struktureeritud lahendamist, kuna sageli hinnatakse programmi korrektsust üksikute komponentide, näiteks funktsioonide kaupa [8]. See lähenemine aitab paremini mõista programmide struktuuri- ja disainipõhimõtteid. Samuti motiveerib automaatne hindamine tudengeid katsetama erinevaid lahendusi, mis soodustab loomingulist mõtlemist ja programmeerimise põhimõtetest sügavama arusaamise teket.

1.2 Programmeerimisülesannete klassifikatsioon

Programmeerimisülesandeid võib klassifitseerida mitmesuguste kriteeriumite alusel.

1.2.1 Tüübi järgi

Simões [9] on välja toonud üheksa erinevat programmeerimisülesannete tüüpi, mida õpilastele lahendamiseks antakse, tuues esile iga ülesandetüübi plussid ja miinused. Simões liigitas ülesanded järgmiselt: „koodi nullist“, „koodi skelett“, „täida lüngad“, „koodi baas“, „leia viga“, „vigane kood“, „kompileerimise vead“, „koodi tõlgendamine“ ja „võtmesõna kasutus“. Lisaks analüüsis Simões, kuidas iga programmeerimisülesande tüüp aitab konkreetseid oskusi arendada ja erinevaid õppe-eesmärke toetada. „Koodi nullist“ ülesanded aitavad õpilastel arendada loovust ja probleemilahendusoskust, samas võivad need olla algajatele rasked, kuna puudub struktuur, millest alustada. Selle probleemi leevendamiseks võib kasutada „koodi skeleti“ ülesannet, kus tudengile antakse ette osaliselt valmis kood, mis võimaldab tal keskenduda konkreetse funktsionaalsuse realiseerimisele. Siin vaatlesime ainult kahte ülesandetüüpi, sest need on kõige olulisemad TÜ kursusel „Programmeerimine“.

Samas Ruf [10] klassifitseeris ülesanded 11 erinevasse tüüpi selle järgi, milliseid on nende lahendamiseks vajalikud oskused. Tema liigitas ülesanded järgmiselt: „kirjuta kood“, „kirjuta etteantud koodi järgi“, „kohanda/laienda/lõpeta etteantud kood“, „optimeeri kood“, „avasta/paranda koodis vead“, „sea etteantud koodile eeldused“, „testi koodi“, „muuda koodi“, „selgita etteantud koodi“, „kirjelda koodi tööd“ ja „joonista koodi diagramm“. Rufi välja toodud ülesandetüüpide hulgas on mitmeid, mis kattuvad Simõesi [9] kirjeldatud tüüpidega (tabel 1). Mõlema autori klassifikatsioon rõhutab programmeerimisülesannete keskseid eesmärke, nagu loovuse arendamine, vigade leidmine ja koodi struktuuri mõistmine. Selline kattuvus näitab, et erinevad uurijad on jõudnud sarnaste järeldusteni, mis

viitab nende ülesandetüüpide olulisusele programmeerimisõppes. Samas pakub Ruf [10] mõnevõrra detailsemat ja täiendavat vaatenurka, näiteks tuues välja ülesandeid, mis keskenduvad teadmiste konkreetsetele esitamisvormidele, nagu diagrammide kasutamine ja probleemide sõnastamine antud koodi põhjal.

Tabel 1. Rufi ja Simõesi programmeerimisülesannete klassifikatsiooni võrdlus.

Simões	Ruf	Selgitus
Koodi nullist	Kirjuta kood	Kirjutada kood algusest lõpuni ise.
Koodi skelett	Kirjuta etteantud koodi järgi	Etteantud koodi järgi kirjutada töötav programm.
Täida lüngad		Täita koodis olevad lüngad, mõistes lahenduse loogikat ilma koodi jooksutamata.
Koodi baas	Optimeeri kood	Parandada töötava koodi loogikat.
	Kohanda/laienda/lõpeta etteantud kood	Muuta etteantud koodi, et lahendus vastaks ülesande nõuetele.
	Muuda koodi	Teisendada etteantud kood uuele kujule või muuta selle esitusviisi.
Võtmesõna kasutus		Kasutada lahenduse kirjutamisel etteantud märksõnu (näiteks konkreetne funktsioon, meetod või programmeerimiskeele element).
Vigane kood	Avasta/paranda koodis vead	Leida etteantud koodist vead ja teha vajalikud parandused.
Leia viga		Leida etteantud koodist vead.
Kompileerimise vead		Analüüsida koodi süntaksivigu ja kompilaatori veateateid ning parandada koodi.
	Testi koodi	Etteantud koodi järgi veenduda selle korrektses toimimises ja tuvastada võimalikke vigu.
Koodi tõlgendamine	Selgita etteantud koodi	Mõista etteantud koodi struktuuri ja selgitada selle tööd.
	Kirjelda koodi tööd	Analüüsida koodi ja välja selgitada, millist probleemi see lahendab.
	Joonista koodi diagramm	Luua etteantud koodile diagramm, mis visualiseerib selle struktuuri, töötamist või loogikat.
	Sea etteantud koodile eeldused	Määrata õiged sisendid või tingimused, et kood käituks ootuspäraselt.

Kursusel „Programmeerimine“ on kõige tüüpilisem ülesandetüüp „koodi nullist“, kuid sellel kursusel on ka muud tüüpi programmeerimisülesandeid – näiteks esimese nädala praktikumi teine ülesanne „Aastaaegade tuvastamine“ [11] on tüüpi „vigane kood“. Selles ülesandes antakse tudengitele programm, mis ei tööta, ning tudengite ülesandeks on tuvastada kõik vead ning need ära parandada. Kuigi sellist tüüpi ülesannet lahendavad tudengid ainult kursuse alguses, tuleb koodi parandamisega ja täiendamisega tegeleda kogu kursuse vältel, sageli automaatkontrollide tagasiside põhjal. Samuti esinevad mitmed muud tabelis 1 väljatoodud programmeerimisülesannete tüübid kursuse ülesannete sees, näiteks peavad tudengid oma koodi optimeerima, iseseisvalt programmi testima ja parandama kompilaatori näidatud süntaksvigu.

1.2.2 Vormi järgi

Selles jaotises on välja toodud kursusel „Programmeerimine“ ülesandetüübi „koodi nullist“ sagedamini lahendavate ülesannete klassid lähtuvalt ülesande vormist. Hiljem analüüsime, mida ja kuidas automaattestid iga ülesandeklassi puhul kontrollima peaksid.

Esimeseks ülesandeklassiks toome välja põhiprogrammi- ehk I/O-tüüpi (Input/Output) ülesanded. Gallo [12] sõnul tähendab I/O „suhtlust sisendseadme (nagu klaviatuur) ja väljundseadme (nagu monitor) vahel“. Kursusel „Programmeerimine“ tähendab põhiprogrammiülesanne harjutust, kus programm saab kasutajalt või failist sisendandmed, töötleb neid ja kuvab seejärel vastava väljundi kas konsoolile või salvestab selle väljundfailidesse. Põhiprogrammiülesande näide on ülesanne „Fizzbuzz“ [13] – kasutajalt küsitakse täisarvu ja seejärel peab programm ekraanile väljastama kas „Fizz“, „Buzz“ või „FizzBuzz“ vastavalt sellele, kas programmile antud täisarv jagub 3-ga, 5-ga või mõlemaga.

Järgmise ülesandeklassi moodustavad funktsioonid, mis on olulised koodi struktuuri ja loogika lihtsustamiseks. Funktsioonid võimaldavad jagada programmi väiksemateks iseseisvateks osadeks ning muudavad koodi selgemaks, paremini hallatavaks ja taaskasutatavaks. Sellist tüüpi ülesannetes peavad tudengid looma funktsioone, mis on korduvalt kasutatavad erinevates olukordades ja aitavad parandada koodi tõhusust ja selgust. Üks näide seda tüüpi ülesandest on „Töötasu“ [14], kus tudengid peavad looma funktsiooni, mis etteantud tundide arvu ja tasumäära puhul peab tagastama töötasu suuruse, ning seda funktsiooni korduvalt kasutama, et leida, milline töötaja saab suurimat töötasu.

Üheks ülesandeklassiks on ka OOP ehk objektorienteeritud programmeerimise ülesanded. OOP tähendab programmeerimisparadigmat, mille aluseks on funktsioonide ja loogika

asemel objektid [15]. Sellist tüüpi ülesandes tuleb luua objektiklass, klassi konstruktor, isendimuutujad ja isendimeetodid, samuti tuleb osata loodud objekti kasutada. Näide sellest ülesandeklassist on harjutus „Isik“ [16], kus tuleb luua klass Isik, klassi konstruktor ja üks isendimeetod, samuti tuleb luua kaks selle klassi isendit ja neid omavahel võrrelda.

Kokkuvõtteks jagunevad ülesanded kolme põhiklassi: põhiprogrammiülesanded, funktsioonid ja objektorienteeritud programmeerimise ülesanded (OOP). Iga klassi puhul tõime konkreetsed näited. Eesmärgiks oli kaardistada ülesandeklassid, mida automaattestid peavad kontrollima.

1.3 Automaatkontrollimise põhimõtted ja strateegiad

Automaatkontrollimise põhimõtted ja strateegiad määravad kindlaks, kuidas erinevate programmeerimisülesannete klasside puhul testimist läbi viia. Järgnevalt kirjeldame automaatkontrolli strateegiaid, mis põhinevad kolmel eelnevalt defineeritud programmeerimisülesannete klassil ning mida tuleks rakendada kursuse „Programmeerimine“ ülesannete hindamisel (vt jaotist 1.2.2).

Ülesannete hindamisel on oluline eristada kahte põhilist testide liiki: käivitavad ehk dünaamilised testid ja staatilised testid [17]. Käivitavad testid keskenduvad lahenduse dünaamilisele käitumisele, hinnates selle reageerimist sisenditele. Staatilised testid seevastu uurivad koodi struktuuri ja loogikat ilma programmi käivitamata.

Põhiprogrammi dünaamilised testid peaksid kontrollima eelkõige väljundi korrektsust. Standardväljundist või väljundfailist saab otsida eeldatavaid elemente, mis võivad olla kas etteantud sõned, arvud, read või muud sarnased üksused ning neid on võimalik väljendada ka regulaaravaldiste abil. Samuti tuleks analüüsida väljundi vormindust, et see vastaks ettenähtud struktuurile, sisaldaks ootuspäraseid elemente ning ei sisaldaks vorminguvigu. Lisaks tuleb kontrollida ka sisendi küsimise õigsust, veendumaks, et sisendandmed on saadud vastavalt ülesandes nõutud küsimismustritele ja spetsifikatsioonidele [17, 18, 19].

Põhiprogrammi staatilised testid keskenduvad koodi sisemise ülesehituse hindamisele ilma programmi käivitamata. Kontrollida tuleks, kas programm pöördub vajalike funktsioonide ja klasside poole, nagu määratud ülesande tekstis. Samuti võimaldavad staatilised testid kontrollida, kas programm sisaldab ettenähtud märksõnu või fraase, kas vajalikud moodulid on imporditud ning kas näiteks tsüklid on kasutusel [17, 18, 19].

Funktsioonipõhiste ülesannete automaatkontroll toimub mitmel tasandil, püüdes tagada lahenduste süntaktilise korrektsuse ja loogilise täpsuse [17]. Funktsiooni dünaamilised

testid käivitavad funktsiooni etteantud parameetritega, et kontrollida tagastusväärtuse ja -tüübi õigsust. Tegelikku tulemust võrreldakse oodatavaga, kasutades kas täpset vastavust või määratud tolerantsi arvuliste väärtuste puhul [19, 20, 21]. Funktsiooni käivituse testides peaks olema võimalik kontrollida ka sisendi küsimise õigsust, ekraaniväljundit ja väljundfailide sisu, analoogiliselt programmi käivituse testidega.

Funktsiooni staatiliste testide käigus analüüsitakse funktsiooni struktuuri funktsiooni käivitamata. Kontrollida tuleks funktsiooni päist, et argumentide arv ja tüübid oleksid õiged. Kontrollida tuleks ka funktsiooni loogilist ülesehitust, sealhulgas teiste funktsioonide kasutamist, vajaliku rekursiooni rakendamist, määratud märksõnade olemasolu ning seda, kas funktsioon on puhas (ei kasuta globaalseid muutujaid) [22, 23].

Objektorienteeritud programmeerimise (OOP) ülesannete automaatkontroll hõlmab klasside ja objektide struktuuri ning käitumise analüüsi. Objektorienteeritud ülesannete dünaamilised testid keskenduvad loodud isendite funktsionaalsuse hindamisele. Testimise käigus tuleks kontrollida, et isenditel oleksid õiged väljad õigete väärtustega, et konstruktor looks spetsifikatsioonile vastavad väljundid (näiteks ekraanile või väljundfailidesse) ning et meetodid toimiksid ootuspäraselt, tagastades või väljastades korrektsed väärtused [24, 25].

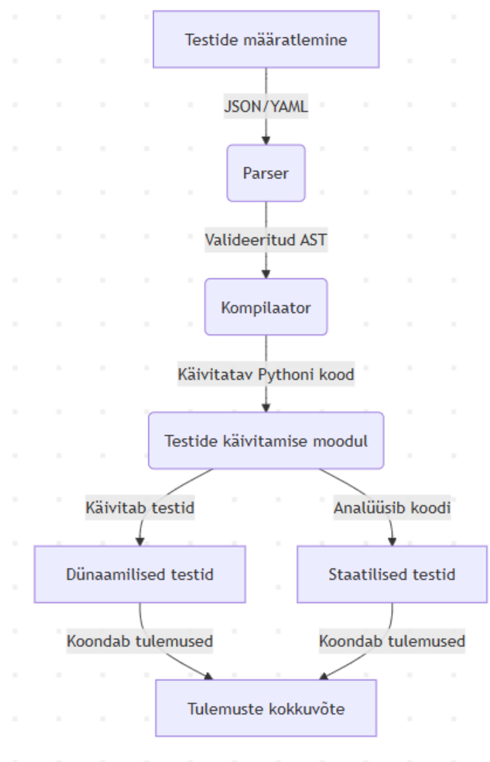
Objektorienteeritud ülesannete staatilised testid analüüsivad klassi struktuuri ja loogikat ilma klassi isendeid loomata. Tuleks kontrollida, kas klass on nõuetekohaselt määratletud, sealhulgas konstruktor, isendimuutujad ja isendimeetodid, kas päriluse mehhanisme on korrektselt rakendatud ning kasutatud on vajalikke meetodeid ja muutujaid [24, 25].

2 TSL

TSL (Test Specific Language) [26] on domeenispetsiifiline automaattestide kirjapanemise keel. Domeenispetsiifilised keeled (DSL) on tarkvarasüsteemides laialdaselt kasutusel, kuna need on loodud kindla ülesande või valdkonna jaoks ning võimaldavad keerukaid tegevusi väljendada lihtsustatud ja struktureeritud viisil [27]. TSL-keele põhiautor on Eerik Muuli kes lõi selle oma doktoritöö käigus.

2.1 TSLi automaatkontrollide töövoog

TSL on toetub JSON- ja YAML-formaadile, mis tagab testide selge ja struktureeritud kirjelduse, tänu millele on neid lihtsam luua ja hallata. Süsteem võimaldab nii dünaamilist testimist kui ka staatilist koodianalüüsi, hinnates ühelt poolt programmi käitumist erinevate sisenditega ning teiselt poolt kontrollides koodi struktuuri ja süntaktilist korrektsust ilma seda käivitamata. Eerik Muuli ja kaasautorite töö kirjeldab, kuidas TSL võimaldab õpetajatel efektiivsemalt hallata programmeerimisülesannete hindamist, vähendades tehtava töö hulka ja tagades objektiivse hindamisprotsessi [26]. Järgnevalt käsitletakse TSLi töövoogu, mis koosneb testide määratlemisest, kompileerimisest, täitmisest ja tulemuste koostamisest (joonis 1).



Joonis 1. TSLi töövoo illustratsioon.

TSL on integreeritud Lahenduse keskkonda [6] ja kogu kirjeldatud töövoog toimub seal. Esmalt toimub testide määratlemine, kus õpetaja või süsteemiadministraator kirjeldab JSON- või YAML-formaadis testide sisu [26] (joonis 2). Ühes failis kirjeldatakse järjest ära kõik programmi hindamiseks vajalikud testid, mille alusel süsteem hakkab lahendust automaatselt kontrollima. Testi kirjelduses määratletakse testitüüp, sisendandmed, eeldatavad väljundid ja hindamiskriteeriumid. See lähenemine võimaldab testimisprotsessi standardiseerida ja muuta selle korduvkasutatavaks, vähendades vajadust käsitsi loodud kontrollimismehhanismide järele.

```yaml
language: python3
validateFiles: true
tslVersion: 1.0.0
requiredFiles:
  - lahendus.py
tests:
- type: program_contains_loop_test
  id: 1
  programContainsLoop:
    mustNotContain: False
    beforeMessage: Kontrollib, kas programmis esineb tsükkel.
    passedMessage: Programmis esines tsükkel.
    failedMessage: Programmis ei esine ühtegi tsüklit.
  name: Programmis esineb tsükkel.
```

Joonis 2. Näide YAML-formaadis TSLi testifailist, mis kontrollib tsükli olemasolu programmis.

Kui testid on määratletud, tuleb need valideerida ja teisendada käivitatavaks Pythoni koodiks. Parser loeb TSL-faili tekstina sisse ja teisendab selle abstraktseks süntaksipuuks, mis koosneb TSL-failis määratletud testiobjektidest [26]. Selle protsessi käigus kontrollitakse faili süntaksit ja loogilist ülesehitust, et tagada testide nõuetekohane struktuur. Kui parser tuvastab vigu, nagu valesti vormistatud testikirjeldused või puuduvad kohustuslikud väljad, väljastatakse veateade ning töövoog katkestatakse. Kui valideerimine õnnestub, edastatakse loodud puustruktuur kompilaatorile.

Kompilaator võtab vastu parseri loodud puustruktuuri ning teisendab selle käivitatavaks Pythoni koodiks [26]. Selle protsessi käigus genereeritakse kood (joonis 3), mis sisaldab vajalikke funktsioonide väljakutseid, mille abil on võimalik määratletud teste käivitada.

```
validate_files(['''lahendus.py'''])
execute_test(
    file_name='lahendus.py',
    generic_checks=[
        {
            'expected_value': True,
            'before_message': 'Kontrollib, kas programmis esineb tsükkel.',
            'passed_message': 'Programmis esines tsükkel.',
            'failed_message': 'Programmis ei esine ühtegi tsüklit.'
        }
    ],
    type='program_contains_loop_test',
    points_weight=1.0,
    id=1,
    name='Programmis esineb tsükkel.',
    inputs=None,
    passed_next=None,
    failed_next=None,
    visible_to_user=True
)
print(Results(None))
```

Joonis 3. Näide Pythoni koodist, mis kontrollib tsükli olemasolu programmis.

Pärast kompileerimist täidetakse testid kahes osas: dünaamilised ja staatilised testid. Dünaamiliste testide puhul käivitatakse tudengi esitatud programm, sisestatakse määratud sisendandmed ning võrreldakse saadud väljundit eeldatava tulemusega [26]. See võimaldab hinnata programmi funktsionaalsust ja õigsust eri stsenaariumides. Staatilised testid seevastu ei nõua programmi käivitamist, vaid analüüsivad koodi struktuuri, kasutades Pythoni abstraktset süntaksipuud (AST) [28]. Selle käigus kontrollitakse, kas programm sisaldab vajalikke funktsioone, tsükleid, klasse ja muid konstruktsioone, mis on ülesande lahendamiseks olulised.

Pärast testide täitmist koostatakse tulemused, mis esitatakse struktureeritud JSON-vormingus (joonis 4). Tulemuste väljund sisaldab informatsiooni iga testi edukuse kohta, võimalikke veateateid ja tagasisidet tudengile [26]. Nende tulemuste abil kuvab Lahendus tudengitele graafilise ja vormindatud tagasiside.

Sellise protsessi abil on programmide hindamine kiire ja objektiivne ning tudengid saavad oma lahendustele ühtlase ja läbipaistva tagasiside.

```json
{
  "producer": "tiivad 0.0.30",
  "result_type": "OK_V3",
  "finished_at": "2025-03-29T11:54:30Z",
  "pre_evaluate_error": null,
  "points": 100,
  "tests": [
    {
      "title": "Programmis esineb tsükkel.",
      "user_inputs": [],
      "created_files": [],
      "converted_submission": null,
      "actual_output": null,
      "actual_file_output": null,
      "exception_message": null,
      "status": "PASS",
      "checks": [
        {
          "title": "Kontrollib, kas programmis esineb tsükkel.",
          "status": "PASS",
          "feedback": "Programmis esines tsükkel."
        }
      ]
    }
  ]
}
```

Joonis 4. Automaatkontrolli tulemuste väljund JSON-formaadis

2.2 TSLi testide struktuur

TSLi testidel on kindel struktuur, mida peab järgima. TSLi abil kirjeldatud testid peavad sisaldama järgmisi parameetreid: testitüüp, testi nimi, unikaalne testi identifikaator, testi eest saadavad punktid, testitüübile omased kohustuslikud väljad ning valikulised, mittekohustuslikud väljad [2].

Igale testile tuleb määrata testitüüp. Joosep Albre [2] koostas oma töös loetelu, kus tõi välja 19 erinevat testitüüpi, mida TSLi abil kirjutada saab. Käesoleva töö kirjutamise hetkeks on testide arv suurenenud 25-ni (vt lisa 1.1). Vahepeal lisandunud kuus testitüüpi keskenduvad objektorienteeritud koodi aspektidele. Need täiendused on lisatud TSLi algsete arendajate, sh Eerik Muuli poolt.

Testil peab olema määratud testi nimi märgisega *name* ja unikaalne identifikaator märgisega *id*. Testi nime kasutatakse tulemuste väljastamisel, et näidata, mida test kontrollib, ning parameeter *id* on oluline, et Lahendus suudaks teste eristada [2].

Punktisumma määramiseks kasutatakse valikulist parameetrit *pointsWeight*, mille vaikeväärtus 1.0 annab testile standardse kaalu koguskooris. Kui *pointsWeight* ületab 1.0, suureneb testi mõju lõpptulemusele, andes testile suurema kaalu. Vastupidisel juhul, kui *pointsWeight* on alla 1.0, väheneb testi mõju ning see test mõjutab lõplikku skoori vähem.

Vastavalt testitüübile sisaldavad erinevad testid nii kohustuslikke kui ka valikulisi välju. Kohustuslikud väljad on vajalikud testi korrektseks toimimiseks ja nende puudumisel tekib kompileerimisviga. Valikulised väljad võib jätta määramata, kuna nende puhul kasutatakse eelmääratud vaikeväärtusi. Näiteks testil *class_instance_test* on kohustuslik parameeter *className*, mis määrab testitava klassi nime, samas valikuline parameeter *dataCategory*, mis määrab väljundandmete tüübi, kasutab vaikeväärtust, kui kasutaja ei määra seda eraldi.

2.3 TSLi edasiarendamise vajadus

TSL on kursuse õppejõudude seas aktiivses kasutuses ja on pakkunud olulist tuge ülesannete kontrollimisel, kuid praktilise töö käigus on ilmnenud mitmeid puudujääke, mis vajavad parandamist. Käesolevas jaotises tuuakse välja TSL-keele edasiarendamise võimalused. Info pärineb kursuse „Programmeerimine“ õppejõududelt, kes kursuse jaoks automaattestide loomisega tegelevad.

Väljundi kontrollimisel on ilmnenud puudujääke. Süsteem ei tuvasta alamsõnesid korrektselt juhul, kui *dataCategory* on CONTAINS_STRINGS ja *checkType* on ALL_OF_THESE. Näiteks, kui kontrollitavateks sõnedeks on „Tuba“ ja „Tuba koristada“, jääb teine sõne tuvastamata, mistõttu testi tulemus on FAIL. Samuti otsitakse praegu väljundist vastust kogu programmi sisend- ja väljundvoost, kuid on vajalik kitsendada otsingut teatud osadele – näiteks enne esimest sisendi küsimist või pärast viimast sisendi küsimist. Lisaks puudub väljundi kontrollimisel regulaaravaldiste tugi, mis muudaks testide kirjutamise paindlikumaks ja lihtsamaks.

Programmist märksõnade otsing on piiratud, kuna *program_contains_keyword_test* ei suuda otsida suvalist sõnet, vaid ainult tähtedest koosnevat sõna. Näiteks failinime „andmed.txt“ tuleb otsida kahes osas, „andmed“ ja „txt“. Selle tõttu tuleb testi kirjutajal koostada ühe testi asemel kaks.

Funktsiooni test *function_is_pure_test* ei tuvasta õigesti, kas funktsioon kasutab ainult lokaalseid muutujaid. Test peaks kontrollima, kas kõik funktsioonis kasutatavad muutujad on defineeritud samas funktsioonis või funktsioonile ette antud argumentidena, kuid praegu see nii ei toimi.

Funktsiooni tagastatud väärtuse kontrollimisel toetutakse praegu vaid ette antud väärtusega võrdumisele, mis ei võimalda hinnata näiteks, kas väärtus kuulub teatud arvude vahemikku, sisaldab konkreetset sõnet või kas järjend, kuigi sisaldab õigeid elemente, on teistsuguses järjekorras. Selle lahendamiseks oleks hea, kui oleks võimalus testis defineerida kohandatud

kontrollifunktsioon, mida rakendatakse tagastatud väärtusele ning mille tulemuse alusel tehakse kontroll.

Samuti puudub võimalus lasta süsteemil automaatselt arvutada funktsiooni tagastatavaid väärtusi – need tuleb eelnevalt eraldi välja arvutada. Oleks lihtsam, kui saaks defineerida funktsiooni, mis arvutab kontrollitava väärtuse, ning seejärel kutsuda seda välja erinevate argumentidega, automatiseerides seeläbi kontrolliprotsessi.

Klassi staatilise analüüsi puhul puudub võimalus kontrollida, kas uuritav klass on teise klassi alam- või ülemklass. Samuti ei võimalda süsteem kontrollida, kas klassi meetod pöördub mõne teise klassi meetodi poole – näiteks kas mõne klassi konstruktor või meetod kutsub välja ülemklassi konstruktorit või meetodit.

Klassi isendi test (*class_instance_test*) ei tunnista väljundi ega väljundfaili kontrolle. Näiteks kui soovitakse kontrollida, kas klassi meetod muudab isendväljade väärtusi ning väljastab midagi, tuleb selleks kirjutada kaks eraldi testi, millest üks on funktsiooni dünaamiline test.

Ülesannete kirjeldustes on sageli nõutud, et funktsioone rakendatakse või klassi isendeid luuakse põhiprogrammis, kuid praegused testid otsivad neid väljakutseid või loomisi ka funktsioonide ja klasside seest. Seetõttu oleks otstarbekas luua uus testitüüp selliselt, et kontrollitakse ainult neid elemente, mis asuvad väljaspool funktsioone ja klasse, st põhiprogrammis.

Testide tulemuste kuvamist tuleks samuti täiustada. Praegu ei ole võimalik näidata kasutajale, milliste argumentidega funktsioon käivitati, milline oli tegelik väljundfaili sisu ning milliste väärtustega olid pärast programmi töö lõppu klassi isendi muutujad. Lisaks, kui soovitakse kasutajale kuvada programmi väljundit, kuvatakse praegu ainult kogu sisend- ja väljundvoog, kuid oleks otstarbekas võimaldada väljundi või selle osade eraldi kuvamist.

Kokkuvõttes on TSL-keele edasiarendamisel vaja parandada väljundi- ja märksõnade kontrolli, funktsioonide ning klasside testimise täpsust ja paindlikkust ning võimaldada kontrollimise ulatuse piiritlemist ainult põhiprogrammiga. Lisaks tuleks täiustada testide tulemuste kuvamist, et saaks anda õppijale detailsemat tagasisidet programmi käitumise ja väljundi kohta.

3 Metoodika

Bakalaureusetöö raames täiustati ja arendati edasi TSL-keelt. Käesolevas peatükis kirjeldatakse tööprotsessi ning meetodeid, mida kasutati keele funktsionaalsuse ja kasutusmugavuse parandamiseks.

3.1 Töökeskkonnad

TSL-keele edasiarendus toimus kahes hoidlas (*repository*). Hoidlas tsl-tiivad [29] arendati Pythoni keeles TSLi mudeli (süsteemi tagasüsteem) loogikat – see moodul vastutab testide käivitamise, valideerimise ja tulemuste genereerimise eest. Selle osa arendamiseks kasutati Visual Studio Code integreeritud programmeerimiskeskkonda (IDE) [30], kuna see pakub head tuge Pythoni keeles programmeerimiseks ning võimaldab süsteemi mugavalt arendada ja testida.

Teine osa tööst toimus hoidlas easy [31], kus asub TSLi kompilaator, mis teisendab TSL-failid Pythoni koodiks. See genereeritud kood edastatakse seejärel moodulile tsl-tiivad edasiseks töötlemiseks ja automaatse hindamise läbiviimiseks. Kompilaator on kirjutatud Kotlini keeles [32] ning selle arendamiseks kasutati IntelliJ IDEA integreeritud programmeerimiskeskkonda, mis on JVM-keelte (nagu Kotlin ja Java) jaoks loodud arenduskeskkond.

Kuna süsteemi olemasolevad komponendid on kirjutatud Pythonis ja Kotlinis, määras see ära ka edasiarendamisel kasutatavad keeled. Mõlemad keeled on kaasaegsed ja laialt kasutusel, mis loob eeldused süsteemi jätkusuutlikuks arendamiseks ja haldamiseks ka tulevikus.

Arendustöö käigus kasutati versioonihalduse süsteemi Git ja selle veebipõhist platvormi GitHub. GitHub võimaldas mugavalt hallata faile ja jälgida tehtud muudatusi, mis lihtsustas arendusprotsessi ning tagas töö käigus tehtud muudatuste selguse ja korrastatuse. Kõik tehtud muudatused lisati harusse tsl_uuendused, kust need seejärel esitati tõmbekutsetena (*pull request*) ülevaatamiseks. Töö käigus loodud tõmbekutsed vaatasid üle TSLi algsed autorid, kes andsid ka vajalikku tagasisidet ja kinnitasid muudatused. See koostöö tagas, et tehtud muudatused on vastavuses süsteemi üldise arhitektuuri ja eesmärkidega.

3.2 Arendusprotsess

Arendustöö algas olemasoleva süsteemi põhjaliku analüüsiga, mille üheks osaks oli tutvumine erinevate testitüüpide eripäradega, mis olid olulised uuenduste kavandamisel. Järgmiseks kaardistati lahendamist nõudvad probleemid, kindlustades, et iga testitüübi

täiustamine või lisamine säilitaks vana süsteemiga tagasiühilduvuse ja ei mõjutaks olemasolevat käitumist. Kaardistatud probleemid on üksikasjalikult esitatud ja analüüsitud jaotises 2.3.

Enne uuenduste teostamist tuli arenduskeskkonnad seadistada. Hoidlas tsl-tiivad on README fail, kus on täpsed juhised, kuidas teste jooksutada. Hoidlas easy tuli projekt üles seada ning siis sai failis DemoApplication TSLi teste kompileerida. Mõlemad eelnevalt nimetatud failid on loodud TSLi algsete arendajate poolt.

Iga uuenduse rakendamisele järgnes testimisprotsess. Esimese sammuna tuli koostada testprogramm, mille käitumist kontrollida. Programmi kontrollimiseks lokaalses arvutis paigutati testprogramm kausta tsl-tiivad. Järgmisena tuli luua kaustas easy TSL-keeles test, mis kompileeriti Pythoni koodiks. Lõpuks kopeeriti kompilaatorist saadud kood kausta tsl-tiivad, kus seda lõpuks jooksutati ja selle tulemusi analüüsiti.

TSL-testide täitmine on kasutusel ka Tartu Ülikooli veebipõhises hindamissüsteemis Lahendus [6], mis võimaldab lahendusi automaatselt hinnata. Kui tudeng laadib oma lahenduse üles, kompileeritakse TSL-failid serveris ning testid käivitatakse isoleeritud Docker-konteineris [26]. Tulemused salvestatakse JSON-vormingus ja esitatakse tudengile veebiliidese kaudu. Käesolevas töös toimus arendamine ja testimine lokaalses keskkonnas, kuna see võimaldas kiiremat katsetamist ilma vajaduseta kasutada olemasolevat pilvekeskkonda või seda eraldi seadistada ja hallata. Arvestades töö mahtu ja eesmärke, osutus selline lähenemine otstarbekamaks.

4 Tulemused

Käesolevas peatükis kirjeldatakse nii olemasolevate testitüüpide täiustusi kui ka töö käigus loodud uusi testitüüpe.

4.1 Olemasolevate testitüüpide parandused

Töö käigus parandati mitmeid olemasolevaid testitüüpe, et suurendada automaatkontrollide täpsust ja töökindlust ning anda õppijatele kiiremat ja üksikasjalikumat tagasisidet.

4.1.1 Väljundi kontroll

Väljundikontrolli algoritmi täiustamiseks muudeti senist lähenemist, et väljundi kontrollimine töötaks korrektselt ka juhul, kui üks sõne on teise alamsõne. Varasem lahendus genereeris kõikide sõnede jaoks üheainsa mustri, mis võis põhjustada olukorra, kus pikem sõne kattus osaliselt lühema sõnega ja jäi seetõttu tuvastamata. Uus lahendus töötleb iga sõnet eraldi, mistõttu otsing on paindlikum ja täpsem.

Uus süsteem võimaldab väljundikontrollis kasutada regulaaravaldisi, kui otsitava sõne alguses on eesliide „regex:“. Regulaaravaldiste abil saab kontrollija määratleda keerukamaid mustreid ja tingimusi ning tänu sellele kontrollida väljundit täpsemalt ja kiiremalt. Näiteks kui on vaja kontrollida, kas programm väljastab mingi täisarvu, siis seda saab kontrollida regulaaravaldise „\d+“ abil. Varem oleks testija pidanud loetlema kõik võimalikud väärtused käsitsi.

Väljundikontrollile lisati uus valikuline väli *outputCategory,* mis määrab, millist osa väljundist kasutaja kontrollida tahab. Selle kategooria võimalikud väärtused on järgmised.

1. ALL_IO – elemente otsitakse programmi kogu sisend-väljundist. See väli on ka vaikeväärtuseks ning enne muudatuste tegemist toimus elementide otsing just selliselt.
2. ALL_OUTPUT – elemente otsitakse programmi kogu väljundist.
3. LAST_OUTPUT – elemente otsitakse programmi kogu väljundist, mis väljastati pärast viimast sisendit.
4. OUTPUT_NUMBER_N – elemente otsitakse programmi kogu väljundist, mis väljastati pärast N-ndat ja enne (N+1)-st sisendit, kus $0 \leq N \leq 9$. Näiteks OUTPUT_NUMBER_0 tähendab, et otsitav skoop on enne esimest sisendi küsimist, OUTPUT_NUMBER_1 tähendab, et otsitav skoop on pärast esimest ja enne teist sisendi küsimist väljastatud tekst jne.

Parameeter *outputCategory* on väljundikontrollis oluline, sest see võimaldab kontrollida programme, mis kasutajaga interaktiivselt suhtlevad. Näiteks saab sedasi kontrollida, kas programmi väljund pärast mingit kindlat sisendit on ootuspärane. See mitte ainult ei paranda testide täpsust ja kiirust, vaid annab kasutajale ka võimaluse täpselt teada, kus tema programm vale väljundi genereeris.

4.1.2 Funktsiooni staatiline test

Testitüübi *function_is_pure,* mis kontrollib, kas funktsioon kasutab ainult lokaalseid muutujaid, parandamiseks tuli analüüsida põhiprogrammis defineeritud muutujaid. Uus algoritm kogub kokku kõik põhiprogrammis defineeritud muutujad ja salvestab nad hulka. Seejärel läbib konkreetse funktsiooni abstraktse süntaksipuu, et leida funktsiooni sees kasutusel olevad muutujad ning salvestab need teise hulka. Kui leitakse kattuvusi kahe eelneva hulga vahel, loetakse test läbikukkunuks, kuna funktsiooni sisu kasutab globaalselt määratletud muutujaid.

4.1.3 Klassi dünaamiline test

Dünaamiline test *class_instance_test* tunnistab nüüd väljundi ja väljundfaili kontrolle, mis tähendab, et testija saab kontrollida, kas klassi konstruktor või meetodid väljastavad või kirjutavad faili oodatud väljundi. Klassi isendi väljundi kontrollimine viiakse läbi funktsiooni ja programmi dünaamiliste testide põhimõtetel, kasutades samu analüütilisi meetodeid väljundite korrektsuse hindamiseks. Tehtud muudatus on kasulik, kui kasutaja tahab samas testis kontrollida, kas klassi konstruktor väljastab õige teksti ning kas klassi isendi väljad on korrektsed pärast isendi loomist. Varem tuli selle funktsionaalsuse testimiseks kirjutada kaks erinevat testi.

4.1.4 Funktsiooni dünaamiline test

Funktsiooni tagastatavate väärtuste kontrollimise loogikat täiustati, lisades võimaluse kasutada hindamiskriteeriumitena lambda-avaldisi ja etteantud funktsioone. Varasemalt lubas süsteem üksnes konkreetse väärtusega võrdlemist, kuid uus lahendus võimaldab dünaamilisemaid kontrollimeetodeid. Kui oodatav väärtus algab sõnega „lambda“ (joonis 5), rakendatakse funktsiooni tagastatud väärtusele vastavat lambda-avaldist, mille tulemusena saadakse tõeväärtus, mis näitab, kas funktsiooni tagastatud väärtus rahuldab määratud tingimusi. Tõeväärtuse *true* puhul loetakse test läbituks ning vastasel juhul läbikukkunuks. Lambda-avaldisega kontroll võimaldab testijal hinnata mitmeid tagastatud väärtuse omadusi, näiteks kas tagastatud sõne pikkus või muster vastab nõutud tingimustele,

```yaml
tests:
- type: function_execution_test
  id: 222
  functionName: test
  functionType: FUNCTION
  arguments: [2,3]
  returnValueCheck:
    returnValue: '''lambda x: 4 <= x <= 7'''
    beforeMessage: Kontrollib, kas funktsiooni tagastatud väärtus on vahemikus 4-7.
    passedMessage: Funktsiooni tagastatud väärtus on õiges vahemiks.
    failedMessage: Funktsiooni tagastatud väärtus ei ole vahemiks 4-7.
  name: Funktsiooni käivituse test.
```

Joonis 5. Näide funktsiooni tagastatud väärtuse kontrollist lambda-avaldisega.

```yaml
tests:
- type: function_execution_test
  id: 223
  functionName: hulkade_otsekorrutis
  functionType: FUNCTION
  arguments:
    - '{"a","b"}'
    - '{1,2,3}'
  returnValueCheck:
    returnValue: |
      '''
      def hulkade_otsekorrutis(hulk1, hulk2):
          tulemus = set()
          for a in hulk1:
            for b in hulk2:
              tulemus.add((a,b))
          return tulemus
      '''
    beforeMessage: Kontrollib, kas funktsioon tagastab õige väärtuse.
    passedMessage: Funktsioon tagastab õige väärtuse.
    failedMessage: Funktsioon ei tagasta õiget väärtust.
  name: Funktsiooni käivituse test.
```

Joonis 6. Näide funktsiooni tagastatud väärtuse kontrollist etteantud funktsiooniga.

kas see on õiget tüüpi, kas tagastatud arv kuulub etteantud vahemikku ning kas tagastatud tulemus rahuldab teatud loogilisi või matemaatilisi seoseid.

Kui oodatav väärtus algab sõnega „def“ (joonis 6), tõlgendatakse seda funktsiooni definitsioonina. Sellisel juhul käivitatakse antud funktsioon etteantud argumentidega lokaalses keskkonnas ning saadud tagastatud väärtus määratleb uue oodatud väärtuse, mida võrreldakse testitava funktsiooni tegeliku tagastatud väärtusega. Kui need kaks väärtust on

samad, loetakse test edukaks; vastasel juhul märgib süsteem testi ebaõnnestunuks. Kui funktsiooni oodatud väärtust tahetakse arvutada etteantud funktsiooni abil, tuleb testi parameetri *returnValue* esimeseks sümboliks panna „|“, et kompileerides ei kaoks ära reavahetused (joonis 6). Niisugune uuendus on kasulik, kuna uuendatud süsteemis ei pea testija õigeid tagastusväärtusi ise arvutama – süsteem teeb selle automaatselt. Kasutaja ülesandeks jääb vaid argumentide sisestamine.

4.1.5 Süsteemi poolt tagastatud tulemused

Süsteemi poolt tagastatud tulemused on TSL-testi täitmise järel loodud struktureeritud andmed, mis sisaldavad teavet sisendi, väljundi, edukuse ja võimalike vigade kohta ning on mõeldud tagasiside andmiseks (joonis 4). Süsteemi tagastatud tulemusi täiendati mitmel viisil, et parandada nende täpsust ja informatiivsust hindamisprotsessi käigus. Tagastatud tulemuste JSON-is on nüüd olemas kogu väljundfaili sisu, sarnaselt tagastatakse ka programmi kogu sisend-väljund. Samuti täiendati kohahoidjasüsteemi, et muuta infoedastus kasutajale efektiivsemaks. Vanas süsteemis oli võimalik kasutada muutujat *expected*, mis näitab, millist väärtust automaatkontroll ootab, ja muutujat *actual,* mis näitab, millise väärtuse automaatkontroll tegelikult sai. Edasiarendatud süsteemis saab kasutajatele edastada järgmised muutujad:

1. *expected* – oodatud väärtus
2. *actual* – tegelik väärtus
3. *function_name* – huvialas oleva funktsiooni nimi
4. *class_name* – huvialas oleva klassi nimi
5. *arguments* – argumendid, millega huvialas olev funktsioon täideti
6. *class_real_fields* – klassi isendi tegelikud väljad ja nende väärtused sõnastikuna pärast programmi täitmist
7. *file_name* – faili nimi, millest elemente otsiti
8. *all_io* – programmi kogu sisend-väljund
9. *all_output* – programmi kogu väljund
10. *last_output* – väljund pärast viimast kasutaja sisendit
11. *output_number_n* – väljund pärast n-ndat ja enne (n+1)-st kasutaja sisendit, kus $0 \leq n \leq 9$
12. *all_file_io* – kogu väljundfaili sisu

Täiendatud kohahoidjate süsteem võimaldab testijal mugavalt õppijale edastada vajaliku info. Samuti saab õppija põhjalikumat teavet selle kohta, mida kontrolliti, kuidas kontrolliti ning millised olid tegelikud tulemused.

4.2 Uued testitüübid

Töö käigus loodi mitmeid uusi testitüüpe, et pakkuda mitmekülgsemat võimalust programmide automaatseks kontrollimiseks (lisa 1.2).

4.2.1 Fraasi olemasolu test

Kuna testitüübid, mille nimes on *contains_keyword*, suudavad programmist otsida ainult tähtedest koosnevaid sõnu, loodi töö käigus uued testitüübid, mille nimes on *contains_phrase* ja mis võimaldavad otsida suvalisi sõnesid ja fraase, sealhulgas ka tühikuid või muid mitte-tähemärke sisaldavaid väljendeid, näiteks „andmed.txt“. Testitüüpides *contains_keyword* kasutati regulaaravaldist \w+, et koguda kokku programmi abstraktse süntaksipuu tekstist kõik sõnad, kuid see lähenemine piirdus ainult lihtsate märgikombinatsioonidega. Uute testide puhul rakendati keerukamat regulaaravaldist (["\'])([^\1]+?)\1|(\w+), mis esmalt otsib jutumärkides märgistatud fraase ning vajadusel sõnade vastavust. Ühtluse mõttes loodi ka uus testitüüp *class_contains_keyword_test*, mis enne puudus.

4.2.2 Klassi päriluse testid

Kaks uut testitüüpi, mis veel loodi, on *class_is_subclass_test* ja *class_is_parentclass_test*. Test *class_is_subclass_test* kontrollib, kas etteantud klass on teise etteantud klassi alamklass. Selle testi algoritm töötab järgmiselt: läbides klassi puustruktuuri, salvestame klassi ülemklassi nime ja hiljem kontrollime, kas see ülemklassi nimi on võrdne oodatava nimega. Test *class_is_parentclass_test* kontrollib, kas etteantud klass on teise või teiste klasside ülemklass. Selle testi algoritm töötab järgmiselt: esmalt läbime kogu programmi abstraktse süntaksipuu tipud ja leiame kõik klasside definitsioonid. Iga leitud klassi puhul kontrollime, kas tema ülemklass vastab etteantud klassile. Kui jah, lisame selle klassi nime hulka, mis hoiab kõiki antud klassi alamklasse. Seejärel rakendame sama kontrolli rekursiivselt äsja leitud klasside puhul, et tuvastada ka nende alamklassid. Lõpuks võrdleme etteantud alamklasside komplekti ja kogutud alamklasside hulka, et määrata testi tulemus. Kui kaks hulka on omavahel võrdsed, loetakse test läbituks, vastasel juhul läbikukkunuks.

```yaml
tests:
- type: class_is_parent_class_test
  id: 222
  className: ParentClass
  genericCheck:
    checkType: ALL_OF_THESE
    expectedValue:
      - ChildClass
      - TestClass
    beforeMessage: Kontrollib, kas klass on etteantud klasside ülemklass.
    passedMessage: Klass on etteantud klasside ülemklass.
    failedMessage: Klass ei ole etteantud klasside ülemklass.
  name: Ülemklassi kontroll.
- type: class_is_subclass_test
  id: 223
  className: ChildClass
  genericCheck:
    checkType: ANY_OF_THESE
    expectedValue:
      - ParentClass
      - TestClass
    beforeMessage: Kontrollib, kas klass on etteantud klassi alamklass.
    passedMessage: Klass on etteantud klassi alamklass.
    failedMessage: Klass ei ole etteantud klassi alamklass.
  name: Alamklassi kontroll.
```

Joonis 7. Näide TSLi testidest *class_is_parentclass_test* ja *class_is_subclass_test*.

Joonisel 7 on näide testist *class_is_parentclass_test*, kus kontrollime, kas ChildClass ja TestClass on mõlemad ParentClassi alamklassid. Joonise teine test *class_is_parentclass_test* kontrollib, kas ParentClass või TestClass on ChildClassi ülemklass.

4.2.3 Klassi isendite ja meetodite staatilised testid

Objektidevahelise suhtluse hindamiseks loodi veel neli testitüüpi: *program_calls_class_function_test*, *class_calls_class_function_test*, *function_calls_class_function_test* ning *class_calls_function_test*. Esimesed kolm testi kontrollivad, kas vastav programmikomponent pöördub klassi etteantud meetodite poole. Viimane test kontrollib, kas klass kutsub välja soovitud funktsioonid. Need testid võimaldavad kontrollida, kas vastavad programmi komponendid kasutavad meetodeid ja funktsioone õigesti, et õppija saaks vajadusel parandada oma programmi üldist struktuuri ja loogikat. Näiteks võimaldab testitüüp *class_calls_class_function_test* testida, kas etteantud klass kutsub välja ülemklassi konstruktorit või meetodeid.

4.2.4 Põhiprogrammi testid

Nüüd on võimalik sooritada kontrolle, mis keskenduvad üksnes põhiprogrammi tööle, kasutades hindamisel kogu programmi süntaksipuud, millest on eelnevalt eemaldatud klasside ja funktsioonide definitsioonid. Põhiprogrammi saab testida järgmiste uute testitüüpidega:

1. *mainProgram_calls_class_test*
2. *mainProgram_calls_function_test*
3. *mainProgram_calls_class_function_test*
4. *mainProgram_contains_loop_test*
5. *mainProgram_contains_keyword_test*
6. *mainProgram_contains_phrase_test*

Kõik kuus uut testitüüpi kuuluvad staatiliste testide hulka, mis ei käivita programmi, vaid keskenduvad selle lähtekoodi struktuurile ja sisule. Nende testitüüpide funktsionaalsus oli juba eelnevalt teostatud; erinevused seisnevad ainult uuritava programmiosa ulatuses. Uued testitüübid on kasulikud, kui ülesande kirjelduses on nõutud põhiprogrammi olemasolu. Näiteks on nüüd võimalik kontrollida, kas põhiprogrammis on pöördutud õigete funktsioonide ja klasside poole, mis aitab testida kogu programmi struktureeritust ja funktsionaalsust.

4.3 Uus süsteem

Jaotises 1.2.2 esitatud programmeerimisülesannete klassifikatsioonist lähtuvalt oli süsteemi arendamise põhifookus just põhiprogrammi, funktsioonide ja objektorienteeritud programmeerimise testimise täiustamisel. Need kolm tüüpi on programmeerimise kursusel kesksel kohal ning nende hindamine peab olema täpne, õiglane ja põhjalik. Samuti tuginesid uue süsteemi lahendused jaotises 1.3 välja toodud strateegiatele ja haarasid nii dünaamilisi kui ka staatilisi teste.

Uue süsteemi üks oluline omadus on tagasiühilduvus eelmise versiooniga, mis tagab, et olemasolevad testid töötavad muutmata kujul ka pärast süsteemi uuendamist. Näiteks kompilaatori moodulisse lisatud uus parameeter *outputCategory*, mis võimaldab täpsemat väljundikontrolli, võiks teoreetiliselt mõjutada testide käitumist. Praktikas see siiski ei juhtu, kuna vaikimisi on parameetri väärtuseks määratud ALL_IO. See tagab, et varasemad testid,

mis ei ole seda parameetrit kasutanud, käituvad endiselt ootuspäraselt, kontrollides kogu sisend-väljundvoogu.

Laiemas pildis pakub uus süsteem vana ees mitmeid olulisi eeliseid. Esiteks vähendab see õppejõudude ajakulu testide kirjutamisel ning võimaldab anda tudengitele täpsemat tagasisidet. Teiseks loovad täiustatud testimise mehhanismid, näiteks funktsioonide ja klasside põhjalikum kontroll, tugevama aluse tudengite programmeerimisoskuste arendamiseks, pakkudes neile detailsemat ülevaadet koodi struktuurist ja loogikast. Kolmandaks on süsteem paindlikum, võimaldades õppejõududel määratleda spetsiifilisi testimiskriteeriume vastavalt õppe-eesmärkidele.

Kui tehtud muudatused integreerida Lahenduse süsteemi, on võimalik automaatkontrolle koostada selliselt, et kirjutada testifaili sisu Lahenduse automaatkontrollide aknas TSLi tekstiväljale. Samas ei ole võimalik luua teste, mis loodi selle töö käigus ja samuti mitmeid seniseid teste, Lahenduse enda testide loomise kasutajaliideses, kus saab teste koostada menüüde ja tekstiväljade abil. Selleks tuleks Lahenduse graafilist liidest täiendada, et oleks võimalik hallata uusi testitüüpe ja nende parameetreid. See eeldab koostööd Lahenduse põhihaldaja Kaspar Papliga.

5 Testimine

Uue süsteemi funktsionaalsuse ja töökindluse tagamiseks viidi pärast iga rakendatud uuendust läbi põhjalik testimine (vt jaotist 3.2). Esmalt testiti süsteemi üksikuid komponente ning seejärel kontrolliti süsteemi terviklikkust, täiendades selleks kursuse „Programmeerimine“ harjutuste teste, mis asuvad Lahenduse keskkonnas. Kokku koostati 20 TSLi testifaili, et kontrollida ja illustreerida tehtud uuenduste funktsionaalsust. Lisaks täiendati kolme olemasolevat testifaili Lahenduse keskkonnast, millest kaks olid kontrolltööde näidisülesanded ja üks objektorienteeritud programmeerimise harjutusülesanne. Kõik töö käigus loodud testifailid on leitavad hoidlas easy.

Üheks ülesandeks, mille teste täiendati, oli kontrolltöö KT2 (variant B) näidisülesanne 2. „Püropood“ [33]. Selle ülesande puhul peab põhiprogramm pöörduma kindla funktsiooni poole ning tsükli sees kasutajalt sisendeid küsima ja nende põhjal korrektsed väljundid genereerima. Täiendatud testid kontrollisid, kas põhiprogramm sisaldab tsüklit ning kas seal kutsutakse välja õige funktsioon. Lisaks hinnati väljundi korrektsust parameetri *outputCategory* abil. Testid koostati nii, et oodatud väärtust otsiti väljundist, mis tekkis pärast n-ndat ja enne (n+1)-st kasutaja sisendit. Vana süsteem ei võimaldanud sellist sisendipõhist vaheetappide kontrolli – väärtusi sai otsida ainult kogu programmi lõplikust väljundist, mistõttu puudus võimalus seostada tulemusi konkreetsete sisenditega. Uus lähenemine võimaldab hinnata programmi tööd kogu töötsükli vältel, mitte ainult lõppvastuse põhjal ning pakub tudengile täpsemat tagasisidet, näidates selgelt, millise sisendi järel viga tekkis.

Süsteemi testimise käigus ilmnes kaks probleemi. Esimene probleem puudutas väljundikontrolli uut algoritmi: kui programmi väljundist püüti leida sama sõne mitmekordseid esinemisi, ei suutnud uus süsteem neid kõiki korrektselt tuvastada. See viga parandati väljundikontrolli algoritmi muutmisega, et tagada korduvate sõnede korrektne tuvastamine.

Teine probleem esines funktsiooni tagastatud väärtuse kontrollimisel. Kui kontrolliti funktsiooni tagastatud väärtust etteantud funktsiooni definitsiooniga ning test oli esitatud JSON-formaadis, siis ei säilinud funktsiooni reavahetused ning testi jooksutamisel tekkis viga. Probleem tekib sellest, et JSON ei toeta mitmerealisi sõnesid samal viisil kui YAML, kus reavahetuste säilitamiseks kasutatakse sümbolit „|”. See probleem ei ole veel lõplikult lahendatud, kuid selle ületamiseks on kaks võimalikku lahendust. Esiteks saab funktsiooni teksti töödelda Python-funktsiooniga json.dumps(), mis teisendab mitmerealise sõne

üherealiseks, võimaldades testi korrektselt jooksutada. Teiseks on võimalik täiendada Lahenduse keskkonda nii, et see teisendaks funktsiooni automaatselt üherealiseks sõneks, vältides seeläbi testi jooksutamisel tekkivaid vigu.

6 Edasiarenduse võimalused

Jaotises 2.3 välja toodud probleemid said käesoleva töö raames lahendatud, kuid süsteemi edasiarendamise ja testide kirjutamise käigus ilmnesid potentsiaalsed suunad edasiseks täiustamiseks. Süsteemi saaks täiendada veel järgmiselt.

1. Kuna selle töö fookus ei olnud Lahenduse enda testide loomise kasutajaliides, tuleks ka sinna kõik selle töö raames tehtud muudatused rakendada. Vaatamata sellele, et töö fookuses ei olnud kasutajaliides, on seda arvesse võetud kogu töö käigus. Autori arvates on see edasiarendus kõige olulisem ja keerulisem, kuna tulevikus peaks Lahendus olema ühtne süsteem, kus saab kõiki teste koostada vastava kasutajaliidese abil.
2. Hetkel ei ole võimalik kontrollida kommentaaride olemasolu. Süsteemi saaks täiendada nii, et oleks võimalik hinnata, kas programm, funktsioon või klass sisaldab nõutud dokumentatsiooni, näiteks kommentaare või päiseid. See võimaldaks õpetajal kontrollida mitte ainult programmi töötamise korrektsust, vaid ka selle loetavust.
3. Praegune süsteem toetab ainult ühest failist koosnevaid lahendusi, mis on piisav kursusel „Programmeerimine“ lahendatavate ülesannete kontrollimiseks. Kui aga tulevikus soovitakse süsteemi rakendada ka suuremate, mitmest failist koosnevate projektide hindamiseks, tuleb süsteemi täiendada, et see toetaks ka mitmefaililisi lahendusi.
4. Kontrollide nimed tuleks uuesti läbi mõelda. Praegu on olemas kontroll nimega *genericChecks*, mis tegelikult kontrollib standardväljundit. Selle nimi võiks olla *standardOutputChecks*. Lisaks on olemas teine sarnase nimega kontroll *genericCheck*, mis läheb sageli segamini kontrolliga *genericChecks*.
5. Staatilistes testides on iga kontroll omaette nimega, näiteks *containsLoop*, *containsReturn*, *isRecursive* jne. Nende kontrollide nimed võiksid olla samad, sest kontrolli tüübi määrab juba ära testitüüp, mille sees nad asuvad.

7 Kokkuvõte

Bakalaureusetöö raames eraldati välja kursusel „Programmeerimine“ kõige sagedamini lahendatavate ülesannete klassid ning kõigi puhul täpsustati, milliseid aspekte tuleks esitatud programmi puhul automaatselt kontrollida. Selle analüüsi ning kursuse õppejõudude kommentaaride põhjal uuendati TSL-keelt nii, et see võimaldaks täpsemat ja paindlikumat automaatkontrolli.

Töö eesmärk – TSL-keele täiustamine programmeerimisülesannete täpsema hindamise võimaldamiseks – sai täidetud. Käesoleva töö käigus parandati olemasolevaid testitüüpe ning loodi 16 uut, mis võimaldavad hinnata varasemast keerukamaid ja mitmekesisemaid ülesandeid. Õppija vaates paranes tagasiside täpsus, mis aitab tudengitel kiiremini tuvastada ja parandada oma lahendustes esinevaid vigu. Täpsem tagasiside võimaldab tudengil paremini mõista, millises osas programm ebaõnnestus, ning toetab seeläbi oskust iseseisvalt vigu analüüsida ja parandada. Õppejõu ja testide koostaja seisukohalt muutus testide loomine efektiivsemaks, mitmekesisemaks ning senisest paremini kohandatavaks erinevat tüüpi ülesannetele. See tähendab, et testide koostamiseks kulub vähem aega ning on võimalus tudengite esitatud lahendusi täpsemalt testida.

Töö käigus toodi välja veel edasiarenduse võimalusi. Kõige olulisem on, et tehtud muudatused süsteemis rakendatakse ka Lahenduse testide loomise graafilisse liidesesse, et uute testide kirjutamine oleks kiire ja intuitiivne.

8 Viidatud kirjandus

[1] Araujo, J., Kim, C. Automated grading systems. 2021 https://www.oxjournal.org/automated-grading/ (10.11.2024)

[2] Albre, J. Tartu Ülikooli kursuse „Programmeerimine" ülesannete automaatkontrollide uuendamine ja üleviimine keskkonda lahendus.ut.ee. 2023. Arvutiteaduse instituut - lõputööde register (10.11.2024)

[3] Tartu Ülikooli õppeinfosüsteem. Kursus „Programmeerimine". ÕIS II (17.11.2024)

[4] Cheang, B., Kurnia, A., Lim, A., Oon, W.-C. On automated grading of programming assignments in an academic institution. Computers & Education, 41, pp. 121–131. 2003. https://doi.org/10.1016/S0360-1315(03)00030-7 (10.11.2024)

[5] Stojanović, T., Lazarević, S. D. Automated grading assignments in programming – advantages, problems, and effects on learning. 2021 https://ebt.rs/journals/index.php/conf-proc/article/view/77 (10.11.2024)

[6] Automaatkontrollide keskkond Lahendus. https://lahendus.ut.ee/ (24.03.2025)

[7] Suleman, H. Automatic marking with Sakai. In Proceedings of the 2008 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on IT Research in Developing Countries: Riding the Wave of Technology, pp. 229–236. 2008. https://doi.org/10.1145/1456659.1456686 (14.04.2025)

[8] Helmick, M. T. Interface-based programming assignments and automatic grading of java programs. In Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education, pp. 63–67. 2007. https://doi.org/10.1145/1268784.1268805 (14.04.2025)

[9] Simões, A., Queirós, R. On the Nature of Programming Exercises. 2020. https://doi.org/10.48550/arXiv.2006.14476 (17.11.2024)

[10] Ruf, A., Berges, M.-P., Hubwieser, P. Classification of Programming Tasks According to Required Skills and Knowledge Representation. In Brodnik A, Vahrenhold J (Eds.), Informatics in Schools. Curricula, Competences, and Competitions, pp. 57–68. 2015. Heidelberg: Springer. https://doi.org/10.1007/978-3-319-25396-1_6 (17.11.2024)

[11] Kursuse „Programmeerimine" esimese nädala praktikum. 2024. Programmeerimine - Kursused - Arvutiteaduse instituut (17.11.2024)

[12] Gallo, K. What is I/O (Input/Output). 2024. https://builtin.com/hardware/i-o-input-output (17.11.2024)

[13] Kursuse „Programmeerimine“ teise nädala praktikum. 2024. Programmeerimine - Kursused - Arvutiteaduse instituut (30.11.2024)

[14] Kursuse „Programmeerimine“ kolmanda nädala praktikum. 2024. Programmeerimine - Kursused - Arvutiteaduse instituut (30.11.2024)

[15] Akiti koduleht. 2024 https://akit.cyber.ee/term/16603 (30.11.2024)

[16] Kursuse „Programmeerimine“ 13. nädala praktikum. 2024. Programmeerimine - Kursused - Arvutiteaduse instituut (30.11.2024)

[17] Dorothy, G., Veenendaal, E.v., Evans, I., Black, R. Foundation of Software Testing. 2006. https://www.utcluj.ro/media/page_document/78/ (04.03.2025)

[18] Fonte, D., da Cruz, D., Gançarski, A. L., Henriques, P. R. A Flexible Dynamic System for Automatic Grading of Programming Exercises. 2nd Symposium on Languages, Applications and Technologies. Open Access Series in Informatics (OASIcs), Vol. 29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 129–144. 2013. https://doi.org/10.4230/OASIcs.SLATE.2013.129 (04.03.2025)

[19] Ala-Mutka, K. M. A Survey of Automated Assessment Approaches for Programming Assignments. Computer Science Education, Vol. 15, No. 2, pp. 83–102. 2005. https://doi.org/10.1080/08993400500150747 (04.03.2025)

[20] Combéfis, S., de Moffarts, G. Automated Generation of Computer Graded Unit Testing-Based Programming Assessments for Education. Computer Science & Information Technology (CS & IT), pp. 91–100. 2019. https://doi.org/10.5121/csit.2019.91308 (04.03.2025)

[21] Python Software Foundation. unittest — Unit testing framework. Python 3.13.2 Documentation. https://docs.python.org/3/library/unittest.html#unittest (05.03.2025)

[22] Møller, A., Schwartzbach, M. I. Static Program Analysis. 2024. https://cs.au.dk/~amoeller/spa/spa.pdf (6.03.2025)

[23] Saikkonen, R., Malmi, L., Korhonen, A. Fully Automatic Assessment of Programming Exercises. In Proceedings of the 6th Annual Conference on Innovation and Technology in

Computer Science Education, pp. 133–136. 2001. https://doi.org/10.1145/377435.377666 (06.03.2025)

[24] GeeksForGeeks, Object Oriented Testing in Software Testing. 2024. https://www.geeksforgeeks.org/object-oriented-testing-in-software-testing/ (06.03.2025)

[25] Tsai, B.-Y., Stobart, S., Parrington, N. A Method for Automatic Class Testing (MACT) Object-Oriented Programs Using A State-based Testing Method. In Proceedings of the Fifth European Conference on Software Testing, Analysis & Review, Edinburgh, 1997. https://citeseerx.ist.psu.edu/document?doi=a7b23214cb3b10bd5fa51a21ad0015bb1b8c62f2 (06.03.2025)

[26] Muuli, E., Palm, R., Lepp, M. Simplifying the creation and maintenance of automated assessments of programming tasks via Test Specific Language. In Proceedings of the 2022 6th International Conference on Education and E-Learning, pp. 14–20. 2023. https://doi.org/10.1145/3578837.3578840 (07.03.2025)

[27] Mernik, M., Heering, J., Sloane, A. M. When and How to Develop Domain-Specific Languages. ACM Computing Surveys, Vol. 37, No 4, 316–344. 2005. https://doi.org/10.1145/1118890.1118892 (08.03.2025)

[28] Python Software Foundation. ast — Abstract Syntax Trees Python 3.13.2 Documentation. 2025. https://docs.python.org/3/library/ast.html (08.03.2025)

[29] TSL-keele mudeli loogika keeles Python. https://github.com/emuuli/tsl-tiivad (09.03.2025)

[30] Microsoft. Python in Visual Studio Code. 2025. https://code.visualstudio.com/docs/languages/python (28.04.2025)

[31] TSL-keele kompilaatori loogika keeles Kotlin. https://github.com/kspar/easy/ (09.03.2025)

[32] Kotlin. IDEs for Kotlin development. 2025. https://kotlinlang.org/docs/kotlin-ide.html (10.03.2025)

[33] Kursuse „Programmeerimine“ kontrolltöö 2 punktilise osa ülesannete näited. 2024. Programmeerimine - Kursused - Arvutiteaduse instituut (30.03.2025)

Lisad

Lisa 1 TSLis kasutatavad testitüübid

Lisa 1.1 Senised testitüübid

1. *program_execution_test* – Kontrollib, kas programm annab etteantud sisenditega käivitades soovitud väljundi.
2. *program_imports_module_test* – Kontrollib, kas programm impordib mooduli(d).
3. *program_calls_function_test* – Kontrollib, kas programm kutsub välja funktsiooni(d).
4. *program_contains_loop_test* – Kontrollib, kas programmis sisaldub tsükkel.
5. *program_contains_try_except_test* – Kontrollib, kas programmis sisaldub try/except plokk.
6. *program_calls_print_test* – Kontrollib, kas programm kutsub välja käsu „print“.
7. *program_defines_function_test* – Kontrollib, kas programmis on defineeritud antud funktsioon(id).
8. *program_defines_class_test* – Kontrollib, kas programmis on defineeritud nõutud klass(id).
9. *program_contains_keyword_test* – Kontrollib, kas programmis sisaldub märksõna / sisalduvad märksõnad.
10. *program_calls_class_test* – Kontrollib, kas programm kutsub välja nõutud klassi isendi / klasside isendid.
11. *function_execution_test* – Kontrollib, kas funktsioon annab etteantud sisenditega käivitades soovitud väljundi.
12. *function_imports_module_test* – Kontrollib, kas funktsioon impordib mooduli(d).
13. *function_calls_function_test* – Kontrollib, kas funktsioon kutsub välja funktsiooni(d).
14. *function_contains_loop_test* – Kontrollib, kas funktsioonis sisaldub tsükkel.
15. *function_contains_try_except_test* – Kontrollib, kas funktsioonis sisaldub try/except plokk.
16. *function_calls_print_test* – Kontrollib, kas funktsioon kutsub välja käsu „print“.
17. *function_defines_function_test* – Kontrollib, kas funktsioonis on defineeritud antud funktsioon(id).
18. *function_contains_keyword_test* – Kontrollib, kas funktsioonis sisaldub märksõna / sisalduvad märksõnad.

19. *function_is_pure_test* – Kontrollib, kas funktsioon kasutab ainult lokaalseid muutujaid.
20. *function_is_recursive_test* – Kontrollib, kas funktsioon on rekursiivne.
21. *function_contains_return_test* – Kontrollib, kas funktsioonis sisaldub käsk „return".
22. *class_instance_test* – Dünaamiline test, mis hindab klassi isendi väljade õigsust.
23. *class_imports_module_test* – Kontrollib, kas klass impordib mooduli(d).
24. *class_defines_function_test* – Kontrollib, kas klassis on defineeritud antud funktsioon(id).
25. *class_calls_class_test* – Kontrollib, kas klass kutsub välja nõutud klassi isendi / klasside isendid.

Lisa 1.2 Töö käigus loodud uued testitüübid

1. *program_contains_phrase_test* – Kontrollib, kas programmis sisaldub suvaline fraas / sisalduvad suvalised fraasid.
2. *program_calls_class_function_test* – Kontrollib, kas programm kutsub välja klassi nõutud meetodid.
3. *function_contains_phrase_test* – Kontrollib, kas funktsioonis sisaldub suvaline fraas / sisalduvad suvalised fraasid.
4. *function_calls_class_function_test* – Kontrollib, kas funktsioon kutsub välja klassi nõutud meetodid.
5. *class_contains_phrase_test* – Kontrollib, kas klassis sisaldub suvaline fraas / sisalduvad suvalised fraasid.
6. *class_is_parentclass_test* – Kontrollib, kas klass on teise klassi / teiste klasside ülemklass.
7. *class_is_subclass_test* – Kontrollib, kas klass on teise klassi alamklass.
8. *class_calls_function_test* – Kontrollib, kas klass kutsub välja funktsiooni(d).
9. *class_calls_class_function_test* – Kontrollib, kas klass kutsub välja klassi nõutud meetodid.
10. *class_contains_keyword_test* – Kontrollib, kas klassis sisaldub märksõna / sisalduvad märksõnad.
11. *mainProgram_calls_class_test* – Kontrollib, kas põhiprogramm kutsub välja klassi(d).
12. *mainProgram_calls_class_function_test* – Kontrollib, kas põhiprogramm kutsub välja klassi nõutud meetodid.

13. *mainProgram_calls_function_test* – Kontrollib, kas põhiprogramm kutsub välja funktsiooni(d).
14. *mainProgram_contains_loop_test* – Kontrollib, kas põhiprogrammis sisaldub tsükkel.
15. *mainProgram_contains_keyword_test* – Kontrollib, kas põhiprogrammis sisaldub märksõna / sisalduvad märksõnad.
16. *mainProgram_contains_phrase_test* – Kontrollib, kas põhiprogrammis sisaldub suvaline fraas / sisalduvad suvalised fraasid.

Lisa 2 Valminud tarkvara

Selle töö käigus tehtud muudatused on kahes hoidlas, milles mõlemas on loodud haru tsl_uuendused, kus asuvad kõik töö raames tehtud täiendused.

1. TSL-keele mudeli loogika keeles Python.
   a. https://github.com/emuuli/tsl-tiivad/pull/2
   b. https://github.com/emuuli/tsl-tiivad/pull/4
2. TSL-keele kompilaatori loogika keeles Kotlin.
   a. https://github.com/kspar/easy/pull/31

Litsents

Lihtlitsents lõputöö reprodutseerimiseks ja üldsusele kättesaadavaks tegemiseks

Mina, Karmo Saviauk

1. annan Tartu Ülikoolile tasuta loa (lihtlitsentsi) minu loodud teos **Automaattestide kirjapanemise keele TSL edasiarendamine**, mille juhendaja on Reimo Palm, reprodutseerimiseks eesmärgiga seda säilitada, sealhulgas lisada digitaalarhiivi DSpace kuni autoriõiguse kehtivuse lõppemiseni.

2. Annan Tartu Ülikoolile loa teha punktis 1 nimetatud teos üldsusele kättesaadavaks Tartu Ülikooli veebikeskkonna, sealhulgas digitaalarhiivi DSpace kaudu Creative Commonsi litsentsiga CC BY NC ND 3.0, mis lubab autorile viidates teost reprodutseerida, levitada ja üldsusele suunata ning keelab luua tuletatud teost ja kasutada teost ärieesmärgil, kuni autoriõiguse kehtivuse lõppemiseni.

3. Olen teadlik, et punktides 1 ja 2 nimetatud õigused jäävad alles ka autorile.

4. Kinnitan, et lihtlitsentsi andmisega ei riku ma teiste isikute intellektuaalomandi ega isikuandmete kaitse õigusaktidest tulenevaid õigusi.

Karmo Saviauk

15.05.2025