Testausdokumentti

Koski-ryhmä

Helsinki 18.5.2004

Ohjelmistotuotantoprojekti

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Kurssi

581260 Ohjelmistotuotantoprojekti (6 ov)

Projektiryhmä

Olli Alm
Seppo Hätönen
Sini Ruohomaa
Antti Takalahti
Sampo Yrjänäinen
Arto Åkerlund

Asiakas

Teemu Kerola

Johtoryhmä

Raine Kauppinen (ohjaaja)
Juha Taina (vastuuhenkilö)
Turjo Tuohiniemi (vastuuhenkilö)

Kotisivu

`http://www.cs.helsinki.fi/group/koski`

Versiohistoria

Versio	Päiväys	Tehdyt muutokset
1.0	10.5.2004	Ensimmäinen versio

Sisältö

2 mega.k91

1 Johdanto

Tämä dokumenti on Koski-projektin tekemän Titokone-konekielisimulaattorin testausdokumentti. Dokumentissa käydään läpi eri testausmenetelmät sekä todetaan mitä niistä käytetään ohjelman testaukseen ja testauksen tulokset. Dokumenttiin liitetään tiedot eri testeistä, joista käy ilmi, mitä testattiin milläkin syötteellä sekä mitä odotettiin testin tulokseksi ja testistä tuli.

Testaus itsessään on prosessi jossa pyritään löytämään ohjelmistosta virheet ja korjaamaan ne valmiiseen tuotteeseen. Testit pyritään laatimaan siten, että ohjelman eri suorituksenvaiheet tulee käytyä läpi mahdollisimman tarkasti.

2 Testauksen kohde ja tavoitteet

Testauksen kohteena on Helsingin yliopiston Tietojenkäsittelytieteen laitoksen Ohjelmistotuotantoprojektin Koski-projektin tekemä Titokone-konekielisimulaattori. Tämä Titokone tulee korvaamaan Tietokoneen toiminta -kurssilla käytetyn Koksi-simulaattorin.

Testauksen tavoitteena on löytää toteutetusta ohjelmistosta virheet ja tarkastaa, että se täyttää vaatimusdokumentissä määritellyt vaatimukset.

3 Testausryhmä ja ympäristö

Testausryhmä koostuu kaikista projektin henkilöistä sekä alfa- ja betatestauksessa kolmesta projektin ulkopuolisesta henkilöstä.

Testaus tehdään Tietojenkäsittelytieteen laitoksen koneilla CSL2 :lla sekä eri windows-ympäristöissä, tarkemmin sekä Windows 2000, että Windows XP -järjestelmissä. Tällä taataan, että toteutettu ohjelma toimii vaadituissa ympäristöissä oikein. Ohjelmointikielenä käytetään CSL2:ssa mukana tulevaa Javan versiota 1.4.2.

4 Testausstrategia

Tässä kappaleessa määritellään eri testauksen vaiheet, jotka ovat luokkatestaus, integraatiotestaus sekä hyväksymistestaus.

4.1 Yksikkötestaus

Yksikkötestauksessa testataan työn alla oleva luokka samalla kun sitä kirjoitetaan. Testaus on pääosin rakenteellista testausta, mutta myös testaan jonkin verran luokan toiminnallisuutta. Yleisimmin virheet löytyvätkin niistä koodin osista, jotka käsittelevät jotain erikoistapausta ja joissa käydään harvoin.

Tämä testaus suoritetaan suurimmaksi osaksi white-box-menetelmällä mutta myös black-box-menetelmällä. Tässä vaiheessa käytetään hyväksi JUnit-ohjelmaa joka on suunniteltu luokkatestaukseen.

JUnit-testejä varten luokille kirjoitetaan tynkäluokkia (stub), joiden toiminta on hyvin yksinkertainen. Niissä on määritelty kaikki oikeiden luokkien metodit mutta ne palauttavat vain jonkin vakion. Näissä luokissa ei ole mitään logiikkaa eivätkä ne kommunikoi keskenään.

White-box menetelmällä pyritään löytämään virheet luokan rakenteista ja metodeista. Optimaalisesti testitapauksilla pyritään käymään läpi kaikki eri loogiset suoritusreitit. Tämä on käytännössä mahdotonta, sillä suorituspolut voivat haarautua useasti, jolloin tutkittavien polkujen määrä kasvaa valtavaksi. Koski-projektin tapauksessa pyritään haarakattavuuteen, eli jokaisessa suorituspolkujen haarassa pyritään käymään ainakin kerran testauksen aikana mutta kaikkia polkuja ei suoriteta. Testitapaukset pyritään luomaan näiden ehtojen pohjalta.

Kun luokkaa testataan, sille luodaan sen tarvitsemat tynkäluokat, jotka palauttavat oikeellisia arvoja, kun luokka kutsuu tyngän jotain metodia. Tällä pyritään saamaan selville, käyttääkö luokka saamiaan tietoja oikein. Luokille tehdään myös JUnit-testejä varten testiluokat. Näihin testiluokkiin kerätään tarpeelliset testit joilla tarkistetaan luokan toimivuus. Nämä testiluokat pyritään säilyttämään, jotta niitä voidaan käyttää mahdollisessa regressiotestauksessa. Niille luokille, jotka sisältävät käytännössä vain metodeja, jotka asettavat arvoja ja palauttavat niitä, ei tehdä JUnit-testejä, vaan ne luetaan tarkasti läpi. Näiden metodien testaaminen on käytännössä turhaa, sillä niissä ei ole loogisia polkuja, jotka voisivat toimia väärin.

4.2 Integraatiotestaus

Integraatiotestauksessa testataan, toimivatko luokat yhteen kuten suunnitteludokumentissa on määritelty. Vaikka kaksi luokkaa toimisivatkin täydellisesti, eivät ne välttämättä toimi keskenään suunnitellulla tavalla. Integraatiotestauksessa pyritään selvittämään luokkien väliset ongelmat ja korjaamaan ne.

Integraatiotestaus voidaan suorittaa periaatteessa kolmella tavalla. Se voidaan suorittaa top-down-menetelmällä, jolloin luokkia liitetään yhteen siten, että integroidaan vaiheittain päätoimintoja. Esimerkiksi koodin kääntämisessä integroidaan ne luokat, jotka ovat tekemisissä koodin kääntämisen kanssa. Jokaisessa testauksen vaiheessa otetaan yksi luokka kerrallaan käsittelyyn paitsi silloin, jos jotain toista luokkaa vaaditaan välttämättä, että luokka toimisi oikein. Kaikista niistä luokista, jotka ovat testattavana olevan luokan alla, kirjoitetaan tynkäluokka, joka korvataan oikealla luokalla testauksen edetessä..

Bottom-up-menetelmässä lähdetään alimmista luokista luokkahierarkiassa, Koski-ryhmän tapauksessa yleensä Info-luokista. Tässä menetelmässä kutakin luokkaa testataan siten, että kun luokan toiminta on varmistettu, otetaan käyttöön sen yläpuolella oleva luokka. Luokista muodostetaan rypäitä joita liitetään lopuksi yhteen.

Näissä kahdessa menetelmässä on kummassakin omat hyötynsä ja haittansa. Top-down-

menetelmässä haittana ovat tynkien määrä ja niiden kirjoittaminen. Hyötynä taasen on se, että päästään hyvin nopeasti testaamaan luokkien kontrollirakenteita. Bottom-up-menetelmässä taasen ei ole tarvetta tyngille ja testitapausten suunnittelu on helpompaa. Haittana taasen on se, että kokonaista ohjelmaa ei ole ennen kuin viimeinen luokka on liitetty. Tästä seuraa se, että jos ylhäällä on huonosti suunniteltu kontrolli, voidaan joutua muuttamaan useita luokkia.

Kolmas tapa on suorittaa testaus yhtä aikaa ylhäältä ja alhaalta. Tällöin voidaan testata ylhäällä olevia kontrollirakenteita samalla kun alhaalla testataan luokkien toimivuutta. Tämä menetelmä voi tuottaa parhaan tuloksen ainakin ajallisesti.

Testaus pyritään suorittamaan lähinnä black-box-menetelmällä sillä tässä vaiheessa luokkien sisäiset rakenteet tulisi olla testattu. Black-box-testauksella pyritään löytämään tässä vaiheessa ne ongelmat, jotka liittyvät siihen, mitä kasatuista luokista tulee ulos, onko niissä puutteellisia rajapintoja jne.

Testauksen edetessä ja todennäköisten virheiden ilmetessä varsinkin pienemmät virheet pyritään korjaamaan suoraan ja suuremmat kontrolloidusti. Tämä johtaa siihen, että kun virheet on korjattu, tulee luokka regressiotestata uudelleen, jotta löydettäisiin mahdollisesti uudet korjauksen yhteydessä syntyneet virheet.

Integraatiotestissä käytetään valmiita testejä, joista kirjataan ylös testin nimi, mitä se testaa milläkin syötteellä ja mikä olisi pitänyt olla tulos sekä tulos testin läpimenosta.

Compilerin ja Processorin testauksessa käytetään valmista ohjelmaa, joka suorittaa kaikki käskyt sekä kaikki muistinosoitukset. Tästä saatuja tuloksia verrataan Koksin antamiin vastaaviin tuloksiin ja tarkistetaan, vastasiko käännetty koodi oikeata koodia ja olivatko prosessorin tilat eri vaiheissa oikeat. Samaa ohjelmaa käytetään myös hyväksymistestauksessa.

4.3 Hyväksymistestaus

Hyväksymistestauksessa kohteena on kokonainen ohjelmisto, jonka tulisi toimia oikein. Tässä vaiheessa käydään läpi vaatimusmäärittelydokumentissa määritellyt vaatimukset läpi ja tarkistetaan, ovatko sekä toiminnalliset että laadulliset vaatimukset täytetty. Tämä testaus käydään läpi black-box menetelmällä käyttäen toteutettuja käyttöliittymiä testaukseen. Tästä testauksesta kerätään puutelista ja sovitaan asiakkaan kanssa jatkotoimenpiteistä.

Alfa-testaus suoritetaan osittain kontrolloidusti siten, että asiakas tai valitut testihenkilöt käyttää ohjelmaa ja Koski-ryhmästä on yksi tai useampi paikalla kirjaamassa havaitut ongelmat ja auttamassa testaajaa.

Beta-testauksessa asiakas ja valitut testihenkilöt testaavat ohjelmaa ja pitävät kirjaa havaitsemistaan ongelmista. Ongelmat voivat johtua todellisista puutteista ohjelmassa tai voivat olla itse asiassa suunniteltuja toimintatapoja.

4.4 Käyttöliittymätestaus

Käyttöliittymää suunniteltaessa pyritään huomioimaan vaatimusmäärittelyssä asetetut vaatimukset. Kun käyttöliittymä on suunniteltu ja siitä on tehty prototyyppi, sitä testataan parilla testihenkilöllä ja korjataan tässä ilmenneet puutteet.

Kun käyttöliittymä on toteutettu, tarkastetaan uudelleen, toteuttaako se esitetyt vaatimukset ja testataan sen toiminnallisuudet. Toiminnallisuutta testattaessa tulee tarkastaa piirtykö käyttöliittymä oikein kun se minimoidaan tai kun sen yli vedetään toisia ikkunoita, pystyykö kaikkea ikkunassa käyttämään, mitä kussakin tilassa pitäisi pystyä, toimivatko kaikki valikot ja niiden alivalikot ja käsittelevätkö kentät niihin syötetyn datan halutulla tavalla.

5 Jatkotoimenpiteet

Löydetyt virheet pyritään mahdollisuuksien mukaan korjaamaan heti, kun ne löydetään. Suuremmat ongelmat jotka vaativat suurempia muutoksia, pyritään korjaamaan kontrolloidusti. Tämä johtaa regressiotestaukseen, jossa pyritään löytämään kaikki muutoksen aiheuttamat ongelmat ja huomioimaan missä ne tapahtuivat. Samalla pyritään miettimään mistä kyseinen virhe johtui ja onko syy kenties jossakin muualla kuin kyseisessä luokassa. Hyväksymistestauksen yhteydessä havaittuja virheitä ei enää korjata, mutta ne kirjataan loppuraporttiin ja korjataan jatkokehityksen yhteydessä.

6 Testit

Tässä kappaleessa käydään läpi eri testit luokittain.

6.1 Yksikkötestit

Yksikkötestejä ei kirjata tässä dokumentissa eritellysti ylös. Tämä johtuu siitä, että JUnit-testejä tehdessä käytettiin tynkäluokkia jotka tekijöiden kokemattomuudesta johtuen eivät osin toimineet toivotusti. Tästä johtuen osa JUnit-testeistä on käyttökelvottomia regressiotestaamisen kannalta. Kuitenkin niillä testeillä, jotka toimivat, huomattiin useita virheitä regressiotestauksessa.

Osalle luokista ei myöskään tehty kattavaa yksikkötestausta vaan ne auditoitiin mahdollisimman tarkasti ja niiden toiminta tarkastettiin integraatiotestauksen yhteydessä. Tämä johtui siitä, että kyseisillä luokilla olisi ollut valtava määrä tynkäluokkia, joiden olisi pitänyt kommunikoida keskenään mikä ei ole tynkien tarkoitus. Luokat GUI sekä GUIBrain testattiin osin yhdessä, sillä niiden toiminta on vahvasti sidoksissa toisiinsa. Näistäkään testeistä ei ole olemassa JUnit-testejä, sillä käyttöliittymälle näitä ei voi tehdä.

Animator ei ollut tärkeällä prioriteetilla toteutusvaiheessa, joten sen toteus alkoi vasta

sitten kun muuta järjestelmää alettiin integraatiotestata ja se valmistui vasta pari päivää ennen demoa. Sitä ei siis ehditty käytännössä yksikkötestaamaan.

6.2 Luokkien yksikkötestauksen kattavuus

Tässä luvussa käydään läpi eri luokkien testien kattavuudet.

6.2.1 Kattavasti testatut

Luokan nimi: Application.java

Testin kattavuus: Kattava.

Luokan nimi: Binary.java

Testin kattavuus: Kattava.

Luokan nimi: BinaryInterpreter.java

Testin kattavuus: Kattava.

Luokan nimi: FileHandler.java

Testin kattavuus: Kattava.

Huomautksia: Ajanpuutteen takia testaus suoritettiin osin integraatiotestauksen yhteydessä. Ks. Luku 6.3.4

Luokan nimi: Loader.java

Testin kattavuus: Kattava.

Luokan nimi: Message.java

Testin kattavuus: Kattava.

Luokan nimi: CompileDebugger.java

Testin kattavuus: Kattava.

Luokan nimi: RunDebugger.java

Testin kattavuus: Kattava.

Luokan nimi: Settings.java

Testin kattavuus: Kattava.

Luokan nimi: SymbolicInterpreter.java

Testin kattavuus: Kattava.

Luokan nimi: Titokone.java

Testin kattavuus: Kattava.

Luokan nimi: Translator.java

Testin kattavuus: Kattava.

6.2.2 Osittain yksikkötestatut luokat

Näitä luokkia ei yksikkötestattu kattavasti. Jokaisen luokan kohdalla on syy minkä takia sitä ei tehty.

Luokan nimi: Animator.java

Testin kattavuus: Osittain.

Syy: Animator valmistui vain pari päivää ennen demoa. Koska sen toiminta on verrattain yksinkertaista, sen testauksessa tarkasteltiin sen piirtämää animaatiota jokaisen käskyn yhteydessä.

Luokan nimi: Interpreter.java

Testin kattavuus: Osittain.

Syy: Interpreter sisältää ainoastaan tiedon käskyjen käännöksista eikä lainkaan toiminnallisuutta. Luokan tiedot on tarkastettu käsin ja integraatiotestauksen yhteydessä. Ks. myös 6.3.6

Luokan nimi: Compiler.java

Testin kattavuus: Osittain.

Syy: Ks. luku 6.3.3

Luokan nimi: Control.java

Testin kattavuus: Osittain.

Syy: Control-luokka on hyvin sidonnainen muihin luokkiin, joten osaa sen metodeista ei pystytty testaamaan kuin integraatiotestauksen yhteydessä tynkäluokkien rajoittuneisuuden vuoksi.

Luokan nimi: Processor.java

Testin kattavuus: Osittain.

Syy: Ks. luku 6.3.5

6.2.3 Yksikkötestaamattomat luokat

Näitä luokkia ei yksikkötestattu JUnit-testeillä, sillä ne sisältävät ainoastaan metodeja, jotka joko asettavat muuttujien arvoja tai palauttavat niitä. Nämä metodit eivät sisällä loogisia rakenteita, jotka voisivat rikkoontua.

Luokan nimi: CompileInfo.java

Luokan nimi: DebugInfo.java

Luokan nimi: LoadInfo.java

Luokan nimi: MemoryLine.java

Luokan nimi: RandomAccessMemory.java

Luokan nimi: Registers.java

Luokan nimi: RunInfo.java

Luokan nimi: Source.java

Luokan nimi: SymbolTable.java

Luokan nimi: Poikkeusluokat

6.3 Integraatiotestaus

6.3.1 Binary-luokan integraatiotestaus

Binary-luokka hoitaa hoitaa käännöksen b91-tiedostosta TTK-91-konekielelle käyttäen hyväkseen BinaryInterpreter-luokkaa. Se hoitaa myöskin jo käännetystä ohjelmasta b91-tiedoston luonnin.

Testin nimi: b91:stä Application-olion luonti

Testin syötteet: b91-tiedosto, joka on valmiiksi String-muodossa, jossa rivinvaihdot on muutettu \n:ksi. Tässä tiedostossa on kaikki komennot, muistiosoitukset, dataa data-alueella sekä symbolitaulussa symboleita ja STDIN- ja STDOUT-määritykset.

Haluttu tulos: Binary palauttaa oikeellisen Application-olion, jota voi käyttää ohjelman ajossa.

Testin tulos: Testi onnistui.

Testin nimi: B91-tiedostossa olevien virheiden havaitseminen.

Testin syötteet: Useita eri b91-tiedostoja String-muodossa, joissa jokaisessa on rikottu formaattia eri tavalla.

Haluttu tulos: Näillä rikkeillä pyritään saamaan kaikki eri virheilmoitukset, jotka ovat mahdollisia b91-tiedostoa luettaessa. Jos jokin virheellinen b91-tiedosto latautui onnistuneesti, testi ei mennyt läpi.

Testin tulos: Testi onnistui.

Testin nimi: Application-oliosta String-esitys.

Testin syötteet: Application-olio, jossa on valmiiksi käännetty ohjelma. Kun Application on syötetty Binary:lle, kutsutaan Binaryn toString()-metodia, jonka tulisi palauttaa String, joka sisältää b91-tiedoston yhtenä String:nä, ja jossa on käytetty rivinvaihtona järjestelmän omaa rivinvaihtoesitystä.

Haluttu tulos: String, jossa on Applicationin sisältämän ohjelman b91-esitys.

Testi tulos: Testi onnistui.

6.3.2 SymbolicInterpreter-luokan yksikkö- ja integraatiotestaus

SymbolicInterpreter toimii kääntäjän apuna erilaisia muunnoksia tehtäessä, ja siksi sen testaaminen oli ensiarvoisen tärkeää. Luokka muuntaa käskyn eri osia merkkijonoista kokonaisluvuiksi. Yksittäisten osien kääntäminen erikseen helpotti myös testausta, sillä rekistereitä oli vain 10 sallittua muotoa (kaksi viimeistä rekisteriä sallittiin kahdessa vaihtoehtoisessa muodossa, esim. FP ja R7 tarkoittavat samaa rekisteriä).

Ainoa suurempi kokonaisuus on koko rivin käännöksen suorittava stringToBinary-metodi joka sekin koostuu yksittäisten osien kääntävistä metodikutsuista. Suurin ongelma oli erikoistapaukset, joissa muistinoutojen määrä vaihteli riippuen muusta käskystä. Esimerkiksi käytettäessä rekisteriä toisena argumenttina täytyy vähentää yksi muistinouto, sillä rekisterin arvoa ei ole määritelty eika siksi voi hakea suoraa osoitetta rekisteristä. LOAD R1, R2 tarkoittaa siis rekisterin kaksi arvon viemistä rekisteriin yksi.

SymbolicInterpreter suorittaa myös merkkijonomuunnokset kokonaisluvuista binääriesitysmuotoon. Tämän toiminta taataan ainoastaan oikeilla parametreillä ja mikäli parametrinä annettu bittien määrä on liian pieni, metodi ei heitä poikkeusta vaan antaa arvon, joka

on väärä. Testit kuitenkin osoittavat, että onnistuneissa tapauksissa metodi antaa oikeita arvoja.

Testit tehtiin niin, että jokainen pienempi metodi testattiin sekä oikeilla että virheellisillä syötteillä.

Testin nimi: Rekisterin arvon palautus kokonaislukuna.

Testin syötteet: Metodia testattiin kaikilla mahdollisilla hyväksyttävillä tavoilla, sekä monella virheellisellä tavalla. Kattavaa virheellisten tapausten testausta ei voitu tehdä jo variaatioiden äärettömän määrän vuoksi, mutta kaikki yleisimmät virheet on testattu ja metodin oikeellisuutta voidaan pitää varmana.

Haluttu tulos: Kokonaisluku nollasta seitsemään.

Testin tulos: Testi suoritettiin onnistuneesti.

Testin nimi: Muistiosoitusmoodin muunnos merkkijonosta kokonaisluvuksi

Testin syötteet: Myös tämä metodi testattiin kattavasti kaikilla onnistuneilla tavoilla, sekä varsin monella virhetilanteella.

Haluttu tulos: -1 kertoo virheestä, onnistunut muunnos on 0, 1 tai 2.

Testin tulos: Testi suoritettiin onnistuneesti, joskin lopussa huomattiin merkkejä kuten 160, joka on non-breaking space, ja ei poistu String-metodilla trim().Tälläiset merkit pitää korvata ennen metodin kutsumista, kuten ohjelma tekee. Metodin toiminta kyseisellä syötteellä on kuitenkin virheellinen.

Testin nimi: Muunnokset kokonaisluvusta 0,1-merkkijonoksi ja takaisin

Testin syötteet: Ohjelmaa testattiin muuttamalla kaikki sallitun arvoalueen kokonaisluvut ensin biteiksi ja sitten takaisin. Lisäksi ohjelmaan syötettiin käsin arvoja arvoalueen päistä ja keskeltä.

Haluttu tulos: Muunnos kokonaisluvusta binääriksi kahden komplementtiesitystä käyttäen.

Testin tulos: Testi suoritettiin onnistuneesti.

6.3.3 Compiler-luokan yksikkö- ja integraatiotestaus

Kääntäjä on todella tärkeä osa prosessia, mutta laajuutensa vuoksi vaikea testattava. Testaus toteutettiinkin osittain automatisoidusti JUnit-testejä käyttäen, osin katselmoimalla ja osittain integraatiovaiheessa ohjelmia ajaen. Ongelmia oli mm. negatiivisten lukujen esityksessä, vaihtelevissa muistinoutojen esityksessa sekä kielen määritelmän kanssa, joka sallii eri osien puuttumisen konekäskyistä, silloin kun niitä ei välttämättä tarvita.

Kääntäjää testattaessa yksikkötestausvaiheessa ongelmia aiheutti toiminnalisuuden puuttuminen. Testit koostuivatkin lähinnä rivin tulkitsijan testaamisesta: parseLine-metodi muuttaa merkkijonon taulukoksi, jossa eri osat ovat omilla paikoillaan. Varsinaista rivin kääntävää metodia ei voinut testata ennen integraatiotestausvaihetta johtuen siitä, että käyttöliittymälle palautetaan CompileInfo-olio, jonka luo CompileDebugger. Vaikeuksia aiheutti myös määritelmä, joka sallii välilyöntejä osien väliin, kuten vaikka muistiosoituksesta kertovan @-merkin, ja varsinaisen osoitteen väliin.

Integraatiotestausta vaikeutti vaihtoehtojen ääretön määrä. Ilman osoiteosaakin vaihtoehtoja tulee huomattavan paljon, ja siksi osa testaamisesta tehtiinkin koodia katselmoimalla ja tiettyihin vaikeisiin tapauksiin keskittymällä.

Kääntäjän testaamista helpotti SymbolicInterpreterin kattava yksikkötestaus ja käännöksessä käytettyjen apumetodien luonne. Kun oli varmistuttu arvojen oikeellisuudesta käskyn eri osien osalta, oli kääntäjän toteuttaminen varsin suoraviivaista, kunhan käsky osattiin purkaa oikein osiin.

Testin nimi: Symbolitaulun päivittäminen.

Testin syötteet: Eri tilanteet, joissa uusia symboleja esitellään: EQU-, DC-, sekä DS-käskyt ja labelin määrittelevä käsky, muuttujaa käyttävä käsky sekä varatun sanan käyttö. Testattiin myös, ettei samaa arvoa lisätä kahta kertaa, vaan vanhaa arvoa päivitetään tarvittaessa.

Haluttu tulos: Symbolitaulu, jossa on vain ja ainoastaan määritellyt symbolit jokainen vain yhteen kertaan. Haluttiin myös oikeanlainen CompileInfo-objekti.

Testin tulos: Testi suoritettiin onnistuneesti.

Testin nimi: Binäärimuodon kääntäminen

Testin syötteet: Ohjelma symbolisessa muodossaan. Testi suoritettiin vertailemalla Titokoneen tulostamaa binäärimuotoa Koksin vastaavaan. Poikkeuksena koneiden käyttämät erilaiset negatiivisen luvun esitykset, josta johtuen koneiden binäärit eroavat osittain toisistaan. Negatiivisten lukujen oikeasta esityksestä varmistuttiin SymbolicInterpreteriä testaamalla.

Haluttu tulos: Yhtenevä binäärimuoto.

Testin tulos: Testi suoritettiin onnistuneesti.

6.3.4 FileHandler-luokan testaus

FileHandlerin eri metodit käyttävät lähes kaikki yhtä ja samaa apumetodia, joten ajan säästämiseksi oikaisimme testaamaan FileHandleria osana muuta järjestelmää.

Testin nimi: Tiedoston lataamisen testaus.

Testin syötteet: Lähdekooditiedosto File-olion edustamana syötetään loadSource-metodille. Tämä kutsuu muiden latausmetodien tapaan loadFileContentsToString()- ja välillisesti loadReaderContentsToString()-metodeja.

Haluttu tulos: Merkkijonoon tiedostosta ladattu lähdekoodi StringBuffer-oliona, jonka oikea muoto voidaan todeta katsomalla sen String-muodon esitystä käyttöliittymässä.

Testin tulos: Testi onnistui.

Testin nimi: Tiedoston tallentamisen testaus.

Testin syötteet: Asetustiedoston merkkijonosisältö syötetään saveSettings()-metodille tallennustiedoston File-olion kanssa. Tämä kutsuu saveStringToFile()-metodia, jota muutkin tiedostotallennusmetodit suoraan käyttävät.

Haluttu tulos: Merkkijonon tarkistetaan päätyneen oikeaan tiedostoon lukemalla ko. tiedosto kyseisen kutsun jälkeen.

Testin tulos: Testi onnistui.

Testin nimi: STDOUT-tiedostoon liittämisen testaus.

Testin syötteet: Avataan STDOUT-tiedosto liittämismuodossa, ja lisätään sinne hieman aloitussisältöä. Ajetaan ohjelma, joka liittää STDOUT-tiedostoon.

Haluttu tulos: Ohjelman mukaan tiedostoon päätyvien tietojen tulee löytyä sieltä luettaessa, ja aloitussyötteen tulee olla paikallaan tiedoston alussa.

Testin tulos: Testi onnistui.

Testin nimi: ResourceBundle-luokan lataamisen testaus.

Testin syötteet: Kutsutaan loadResourceBundle()-metodia neljästi, parametrinä 1) käännöstiedosto, joka ei ole olemassa, 2) käännöstiedosto, jonka sisältämää luokkaa ei voida ladata väärän nimen takia, 3) “käännöstiedosto”, joka sisältää muun kuin ResourceBundle-luokan sekä 4) käyttöliittymän kautta valitaan varsinaisen kelpuutettavan ResourceBundle-luokan sisältävä käännöstiedosto. Tarkkaillaan heitettäviä poikkeuksia.

Haluttu tulos: Kolmen ensimmäisen käännöstiedoston tulee aiheuttaa tietynviestinen poikkeus (viestistä selviää alkuperäispoikkeuksen tyyppi, esim. ClassNotFoundException). Neljännen tiedoston onnistunut valinta aiheuttaa käyttöliittymän kielen vaihtumisen.

Testin tulos: Testi onnistui.

Testin nimi: Tiedostopäätteen muuttamisen testaus.

Testin syötteet: Kutsutaan changeExtension()-metodia tiedosto.k91:stä tiedosto.b91-tiedostoon nimen vaihdon sekä kaannosluokka.class-tiedostosta kaannosluokka-resurssinimeen vaihdon yhteydessä vastaavalla File-oliolla ja halutulla uudella päätteellä (tyhjä merkkijono jälkimmäisessä tapauksessa).

Haluttu tulos: Tuloksena saadun File-olion tiedostopäätteen tulee olla vaihtunut oikein edellä kuvatulla tavalla.

Testin tulos: Testi onnistui.

Testin nimi: InputStreamin lataamisen testaus.

Testin syötteet: Ajetaan Titokone, kun käyttäjällä ei ole vielä omaa asetustiedostoa. Tällöin asetukset luetaan InputStream-muodossa oletusasetustiedostosta, jota ei edusteta muiden tiedostojen tapaan File-oliona jar-paketoimisen mahdollistamiseksi. Muokataan oletusasetustiedosto tätä ennen sisältämään kovakoodatuista asetuksista eroavat muistin koon, suoritus- ja käännösasetukset. (Asetustiedostojen vaillinainen sisältö ei aiheuta ongelmia.)

Haluttu tulos: Asetustiedoston lataamisen onnistuminen voidaan todeta kahdesta eri lähteestä: -vvv-parametrillä käynnistetty Titokone kertoo, että oletusasetustiedosto ladattiin, se sisälsi x riviä ja että siitä löytyi tietty määrä avain- ja arvopareja. Lisäksi koneen muistin koko, ajo- ja käännösasetukset sekä käyttöliittymän kieli voidaan ohjelman käynnistyttyä tarkistaa olevan muokatun tiedoston asetusten mukaiset.

Testin tulos: Testi onnistui.

6.3.5 Processor-luokan yksikkö- ja integraatiotestaus

Processor-luokka on keskeinen projektin osa, sillä se tulkitsee komennot toiminnoiksi. Toiminnan oikeellisuuden arvioimista vaikeutti se, että alkuperäisessä Koksissa esimerkiksi joidenkin komentojen muistinoutojen määrä todettiin Koksia tutkailtaessa muutetuksi alkuperäisestä TTK-91-konekielisen koodin syntaksista eroavaksi. Tällöin luokan todennäköisimmät ongelmat johtuivatkin siitä, että sen ohjelmoijilla ei ollut riittävää tietoa toiminnan yksityiskohdista.

Processor tuntee kaikki TTK-91-konekielen komennot, jotka löytyvät spesifikaatiosta, ja se sisältää lähinnä kahta luokkaa metodeja: osa on hyvin yksinkertaisia muuttujien tallennuksia ja osa sellaisia, joiden lopputulos riippuu lähes täysin Processor-luokan ja sen käyttämien luokkien tilasta, ja on havaittavissa vain näiden luokkien tilan muutoksina. Processor-luokan yksikkötestaus todettiin siinä määrin työlääksi tehtäväksi, että sen sijaan koodi päätettiin katselmoida. Metodijako ja nimeämiskäytäntö Processor-luokassa on siinä määrin onnistunut, että koodin etenemisen seuraaminen oli helppoa. Katselmoinnissa havaittiin muutamia bugeja ja rajoituksia, jotka korjattiin. Puutteet liittyivät odotusten mukaisesti lähinnä eksoottisempaan toimintaan, jolloin osa niistä tarkentuikin toiminnan määrittelyn puutteiksi.

Processor käyttää enimmäkseen varsin yksinkertaisia luokkia. Kun nämä oli yksikkötestattu, Processor kytkettiin niiden kanssa yhteen integraatiotestausta varten. Tässäkin keskeiseksi katsottiin oikeiden tilanmuutosten tapahtumisen varmistus, sillä kaikkien mahdollisten väärien tilanmuutosten tarkistus ei olisi ollut ajan puutteen takia mielekästä.

Testin nimi: Komentojen oikea suoritus: memoryInput() ja runLine().

Testin syötteet: Testissä syötettiin memoryInput-metodin avulla prosessorin muistiin 80-rivinen TTK-91-ohjelma rivi kerrallaan, valmisteltiin Processor ohjelman ajoa varten, tarkistettiin sen rekisterien tila ja ajettiin ohjelma rivi riviltä runLine()-metodin avulla. Ohjelma oli tätä ennen käännetty symbolisesta konekielestä binääriksi Compiler-luokan ja sen apuluokkien avulla.

Ohjelma alkaa testaamalla SVC:tä lukuunottamatta kaikki konekielen käskyt kerran, enimmäkseen parametreillä R1, =1. Hyppykäskyt hyppäävät seuraavalle riville, jolloin niiden hyppyehdon oikea tarkistus jää järjestelmätestauksessa varmistettavaksi. Ohjelman viimeiset 40 riviä testaavat LOAD- ja STORE-käskyn erikoisuuksia ja viimeinen rivi testaa SVC-kutsun parametrein SP, =HALT.

Haluttu tulos: Ohjelman latauksen ja “ajoon valmistautumista” edustavan runInit()-metodin kutsun jälkeen tarkistettiin rekisterien (etenkin SP ja FP) oikeat alkuarvot, sekä varmistettiin Processor-luokan olevan ajotilassa. Kunkin rivin suorituksen jälkeen tarkistettiin PC:n arvo, Processorin tila (tilan tuli loppuun asti olla STATUS_STILL_RUNNING) sekä halutun muistirivin tai rekisterin arvon muutoksen toteutuminen. Esimerkiksi kun rekisterissä R1 oli arvo 100 ja suoritettiin rivi ADD R1, =1, tarkistettiin, oliko rivin suorituksen jälkeen rekisterissä R1 arvo 101.

Testin tulos: Processor suoritti käskyt oikein.

Testin nimi: Ajoaikaisten poikkeusten heittäminen.

Testin syötteet: Kuten komentojen oikeaa suoritusta testattaessa, käännettiin nytkin testiohjelma, joka syötettiin memoryInput-metodin avulla Processor:lle. Tällä kertaa syöte valittiin siten, että testauksen aikana kaikki Processor:n eri TTK91RuntimeExceptionit tulisivat heitetyiksi. Aluksi luotiin Processor-olio, johon oli liitetty vain kymmenrivinen muisti, ja memoryInput()-metodin avulla yritettiin syöttää sinne riviä liian pitkä ohjelma. Tämän jälkeen luotiin kullekin testille oma Processor, johon syötettiin minimaalinen, poikkeuksen aiheuttava ohjelma, ja kutsuttiin ajoon valmistavaa runInit()-metodia.

Haluttu tulos: Sekä liian suurta ohjelmaa ajettaessa että poikkeuksen aiheuttavia rivejä testattaessa, testi läpäistiin vain jos oikea poikkeus heitettiin oikealla hetkellä. Sekä väärä poikkeus että poikkeuksen heittämättä jääminen olisivat aiheuttaneet testin epäonnistumisen.

Testin tulos: Testi onnistui.

6.3.6 GUI-luokkien testaus

Käyttöliittymän testausta on hankala automatisoida, sillä syötteet ja tarkkailtavat tilanmuutokset ovat lähinnä ihmisten tehtäväksi ja havainnoitavaksi suunniteltuja. GUI-luokkia on täten lähinnä järjestelmätestattu, aluksi korvaamalla muu järjestelmä tynkäluokilla ja myöhemmin integroiden. Systemaattista testausta hankaloitti GUI-luokkien jatkuva muutos, joka samalla piti niiden parhaan tuntijan kiireisenä. Jatkuva muutos johtui osittain siitä, että osan ryhmän jäsenten taidot toteuttaa käyttöliittymä Javalla eivät olleet alun perin projektin kunnollisen toteuttamisen kannalta riittävällä tasolla. Luokkiin onkin tullut useita muutoksia myös järjestelmätestauksen aikana johtuen juuri Swing-luokkien toiminnan yllättävyyksistä sekä myös Javan säikeiden käytöstä, josta käyttöliittymän toteuttajalla ei ollut aiempaa kokemusta lainkaan, mikä aiheuttikin huomattavia hankaluuksia. Suurin osa järjestelmätestauksessa havatuista kriittisistä virheistä johtui juuri säikeistä ja niiden rinnakkaisesta suorituksesta, minkä takia jouduttiin kirjoittamaan paljon koodia uusiksi.

Itse GUI-luokkaa yksikkötestattiin tarkistamalla, että käyttöliittymä näyttää ruudulla oikealta ja reagoi syötteisiin oikealla tavalla. GUI ei sisällä juurikaan muuta toiminnallisuutta, joten sen muunlainen (systemaattinen) testaus olisi ollut ylivoimaista. Tämän jälkeen GUI:n rooli olikin auttaa GUIBrainin yksikkötestausta, mikä oli samalla GUI:n ja GUIBrainin keskinäistä integraatiotestausta. Niitä kuitenkin testattiin erillään muista luokista ja apuna käytettiin useita tynkäluokkia, joita muuteltiin sen mukaan millainen testi haluttiin ajaa. Näistä tynkäluokista on valitettavasti vain viimeiset versiot tallella. Periaatteessa pyrimme kuitenkin niiden avulla käymään läpi kaikki mahdollisuudet arvot, joita tynkien sijaamat luokat saattaisivat palauttaa. Koska luokkia oli niin paljon ja mahdollisuuksia lukuisia, joiden lisäksi vielä itse GUIBrain on isokokoinen unohtamatta sitä, että sen metodeja voidaan suorittaa rinnakkain eri säikeissä, niin täysin kattavien testien ajo oli silkka mahdottomuus. Siksipä suurin osa virheistä löytyikin vasta integraatiotestausvaiheessa, jossa tynkäluokista oltiin luovuttu ja saatu käyttöön oikeat luokat. Tällöin erilaisten testien ajo oli vain erilaisten TTK-91-ohjelmien kirjoittelua sekä asetusten muuttelua. Itseasiassa paras tapa testata GUIBrainia olikin miettiä erilaisia tapoja, jolla mahdolliset virhetilanteet tulisivat esille ja suunnitella niille oma testinsä. Tämä menetelmä paljastikin erittäin monta virhettä, mutta niitä varmasti jäi edelleen.

Testin nimi: Binääritiedoston avaaminen

Testin syötteet: Tiedosto jonka pääte on b91.

Haluttu tulos: Mikäli tiedosto on formaatin mukainen binääritiedosto, kyseisen tiedoston sisältämä ohjelma näytetään ruudulla. Jos taas tiedosto ei päätteestään huolimatta ole binääritiedosto, niin näytetään virheilmoitus.

Testin tulos: Testi onnistui.

Testin nimi: Kooditiedoston avaaminen

Testin syötteet: Tiedosto jonka pääte on k91.

Haluttu tulos: Tiedoston sisältö näytetään ruudulla.

Testin tulos: Testi onnistui.

Testin nimi: Kooditiedoston kääntäminen

Testin syötteet: Tiedostoja jotka sisältävät TTK-91-konekielen formaatin mukaisia ja virheellisiä ohjelmia.

Haluttu tulos: K91-muotoisten ohjelmien kääntämisen tulisi edetä siten, että jokaisella kierroksella päivitetään näyttöä ja asetuksista riippuen jäädään odottamaan jatkamiseen käskyttävää syötettä. Käännöksen suoritettua näytöllä tulisi näkyä ohjelma binäärimuodossa. Tilanteessa jossa syötteenä annettiin jokin muu tiedosto, virheellisestä syötteestä tulisi näkyä virheilmoitus.

Testin tulos: Testi onnistui.

Testin nimi: Ohjelman suorittaminen

Testin syötteet: Käännettyja K91-konekielisia ohjelmia, joista jotkut sisältävät virheellisiä käskyjä. Lisäksi annettava tieto siitä, suoritetaanko ohjelma kommentoiden, riveittäin vai animoiden.

Haluttu tulos: Suoritus etenee siten, että näyttöä päivitetään jokaisella kierroksella ja asetuksista riippuen jäädään odottamaan syötettä jatkamiseen. Onnistunut testi päättyy joko virheilmoitukseen, mikäli suoritettavassa ohjelmassa oli jokin virheellinen käsky tms. tai ilmoitukseen siitä, että suoritus päättyi onnistuneesti.

Testin tulos: Testi onnistui.

Erikoistapaus käyttöliittymäluokista on Animator, joka toteuttaa “vapaaehtoisen” vaatimuksen suoritussyklin animoinnista. Animointiin ei rohjettu kiinnittää tekijää ennen kuin jo kunnioitettavan kokoiseksi paisunut järjestelmä oltiin saatu jo pitkälti toteutettua. Animator valmistuikin testausvaiheen ollessa jo pitkällä, ja sitä testattiin lähinnä koodauksen aikana (kunkin komennon toteutuksen jälkeen) ja järjestelmän osana, kun se saatiin muutamaa päivää ennen koodin jäädyttämistä yhdistettyä muuhun järjestelmään. Animatorin toiminta on onneksi myös rinnakkaistoteutuksesta johtuen varsin erillistä muuhun järjestelmään nähden, joten siihen mahdollisesti jääneiden virheiden ei katsottu haittaavan sovelluksen yleistä toimintaa niin paljon, että luokka olisi kannattanut jättää pois kokonaisuudesta. Luokkaa on tämän jälkeen järjestelmätestattu silmämääräisesti, enimmäkseen peruskomennoilla kuten LOAD, STORE ja ALU-operaatioilla. Esimerkiksi IN- ja OUT-komentojen toimintaa ei ehditty muokata mahdollisimman opiskelijaystävällisiksi – muistinhallintayksikön osoitekenttään MAR tulisi tallentaa jotakin laitteeseen viittaavaa, mutta koska samaa kenttää käytetään muistiosoitukseen, tämän tulisi erota selvästi muistiosoitteista.

6.4 Hyväksymistestaus

Ajan puutteen takia kunnollista alfa-testausta ei ehditty suorittaa kuin yhdellä testihenkilöllä. Kuitenkin ryhmän jäsenet testasivat ahkerasti itse eri ohjelmilla ja syötteillä Titokonetta ja suurin osa ongelmista saatiin selville ja korjattua. Beta-testaukseen päästiin kuitenkin niin myöhään, että siinä löytyneitä virheitä ei ehditä enää korjaamaan tämän projektin aikana.

6.4.1 Vaatimusten testaus

Tässä kappaleessa käydään läpi vaatimusmäärittelydokumentissa analysoidut vaatimukset ja niiden paikkansapitävyys titokoneessa.

Vaatimus: V1. Soveltuu Tito-kurssille

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ohjelma testattiin kolmella projektin ulkopuolisella testihenkilöllä ja ohjelma toimii eri käyttöjärjestelmissä sekä mukaan liitetään asennusohjeet sekä käyttöohjeet. Myös käyttöliittymä on suunniteltu niin, että se on yksinkertainen ja selkeä sekä soveltuu videotykille.

Vaatimus: V2. Aiemmat koodit yhteensopivia

Täyttyikö vaatimus: Kyllä

Huomautuksia: Jotkin esimerkkeinä annetuista koodeista eivät noudattaneet annettua spesifikaatiota eivätkä ne toimineet edes Koksi-ohjelmassa sekä joissakin koodeissa oli tarkoituksella jätetty virheitä. Näitä ohjelmia emme tue. Myöskin muutettuamme Koksin kääntämää binääriesitystä, eivät aiemmat binääriesitykset käänny oikein Titokoneessa. Tämä ei riko vaatimusta, sillä aikaisemmin ei ole ollut olemassa b91-tiedostoja.

Vaatimus: V3. Noudattaa Koksia ennemmin kuin spesifikaatiota.

Täyttyikö vaatimus: Osittain

Huomautuksia: On olemassa tapauksia, joissa totesimme, että Koksi ei noudata omaa spesifikaatiotaan. Näissä tapauksissa päätimme noudattaa ennemmin spesifikaatiota. Toisaalta paremmin esitetyillä kysymyksillä olisi voinut saada selkeämpiä vastauksia.

Vaatimus: V4. Toimii sekä Windows- että Linux-koneissa

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ohjelmaa on testattu Windows 2000 ja Windows XP sekä CSL-Linux ja Fedora-ympäristöissä. Lisäksi projektin jäsenet käyttivät kehitykseen edellä mainittuja ympäristöjä.

Vaatimus: V5. Koostuu erillisistä osista

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ohjelma on pyritty osittamaan erillisiin osiin hyvän oliopohjaisen ohjelmoinnin periaatteiden mukaisesti.

Vaatimus: V6. Perusnäkymä näyttää koneen tilan kullakin hetkellä.

Täyttyikö vaatimus: Asiakkaan / Koksin vaatimalla tasolla, kyllä.

Huomautuksia: Koneen tila kattaa rekisterit, muistin jne. Tilarekisteri näytetään vain animaatioikkunassa.

Vaatimus: V7. Erotettavissa oleva koodi- ja data-alue

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Aiemmin aluksi koodi- ja data-alue olivat ensin yhdessä, mutta ne pystyi erottamaan erillisiksi osiksi. Titokoneessa ne on suoraan erotettu toisistaan juoksevaa rivinumerointia jatkaen.

Vaatimus: V8. Ei editointimahdollisuutta simulaattorissa.

Täyttyikö vaatimus: Ei.

Huomautuksia: Ohjelman koodia pystyy muokkaamaan Titokoneessa. Tällä voidaan korjata nopeasti yksittäisiä kirjoitusvirheitä koodissa.

Vaatimus: V9. Koodi englanniksi ja käyttöohje sekä englanniksi että suomeksi

Täyttyikö vaatimus: Kyllä

Huomautuksia: Ei huomautettavaa.

Vaatimus: V10. Luokkien koodi on helposti ymmärrettävää

Täyttyikö vaatimus: Osittain.

Huomautuksia: Suurin osa koodista on helposti ymmärrettävää kommentointeineen, mutta varsinkin monimutkaiset luokat voivat osittain olla vaikeaselkoisia. (Esim. GUI:n luokat, Processor, Compiler.)

Vaatimus: V11. eAssarin tukeminen

Täyttyikö vaatimus: Kyllä

Huomautuksia: EAssaria varten on tehty testiluokka AssariUI, jota ajetaan servletinä.

Vaatimus: V12. Ohjelma tukee itseään muuttavaa koodia

Täyttyikö vaatimus: Lähes.

Huomautuksia: Uudelleenlatauksen yhteydessä käyttöliittymä ei päivity muutettujen rivien osalta, mutta prosessorilla on alkuperäiset rivit käytössä. Ongelma on siis vain Titokoneen käyttöliittymässä.

Vaatimus: V13. Muistissa näytetään myös mahdolliset konekäskyt

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Koko muistin alueelta ei kaikkia rivejä käännetä, vaan pelkästään ne, mitä on ohjelman ajon aikana alustettu tai käytetty.

Vaatimus: V14. Käytössä koko tiedostojärjestelmä

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Javan tiedostodialogi tukee tätä suoraan.

Vaatimus: V15. Muistin koko, kieli ja STDIN- ja STDOUT-tiedostot vaihdettavissa

Täyttyikö vaatimus: Kyllä

Huomautuksia: Näitä voidaan muuttaa titokoneen Asetukset-valikon alta sekä STDIN- ja STDOUT-tiedostoja DEF-käskyllä. Muutokset tallentuvat käyttäjäkohtaiseen asetustiedostoon.

Vaatimus: V16. Käsky SHRA lisätty operaatiokoodilla 27

Täyttyikö vaatimus: Kyllä

Huomautuksia: Ei huomautettavaa.

Vaatimus: V17. Tiedostojen käsittely IN- ja OUT-käskyillä

Täyttyikö vaatimus: Kyllä

Huomautuksia: Ei huomautettavaa.

Vaatimus: V18. Käskyn suorituksen animointi

Täyttyikö vaatimus: Kyllä

Huomautuksia: Animator-luokkaa ei ole testattu tarkasti, mutta se toimii silmämääräisesti oikein.

Vaatimus: V19. Laiteajurin animointi

Täyttyikö vaatimus: Ei.

Huomautuksia: Tästä sovittiin asiakkaan kanssa.

Vaatimus: V20. Ohjelmaa voi käyttää sekä Windows että Linux käyttöjärjestelmissä.

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ks. V4.

Vaatimus: V21. eAssarin kanssa yhteensopiva

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Tätä varten tehtiin www-käyttöliittymä jonka pitäisi mahdollistaa eAssarin käyttö suunnitelluilla rajapinnoilla.

Vaatimus: V22. Ylläpidettävyys

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ks. V5. ja V10.

Vaatimus: V23. Käyttöliittymän joustavuus (muistin koon muuttaminen yms.)

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Asiakkaan pyytämät asetukset toteutettiin.

Vaatimus: V24. Ohjelmiston virheetön toiminta

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Yleisesti testauksen perusteella ohjelmien suoritus on virheetöntä. Loppuraportissa kuvaillaan joitakin Titokoneen puutteita, joita ei kuitenkaan ole katsottu kriittisiksi.

Vaatimus: V25. Toiminnalliset tavoitteet 1-17 sekä laadulliset vaatimukset täyttyvät

Täyttyikö vaatimus: Kyllä

Huomautuksia: Ks. muut tavoitteet.

Vaatimus: V26. Luotettavuus.

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ks. V24.

Vaatimus: V27. Ohjelman tehokkuus.

Täyttyikö vaatimus: Kyllä

Huomautuksia: Ohjelma ei pääse tavoitteeksi asetettuun viiteen sekuntiin (n. 13 s ensimmäisellä käynnistyksellä, 8 s jatkossa käyttöjärjestelmästä riippumatta) Porkkala-luokassa (PIII-400/256MB), mutta Stubben-luokassa käynnistykseen meni noin kuusi sekuntia.

Vaatimus: V28. Ohjelmiston itsesuojelukyky

Täyttyikö vaatimus: Suurimmaksi osaksi.

Huomautuksia: Joitakin luokkia voi “väärinkäyttää”; luokka-castauksen avulla esimerkiksi TTK91Memoryyn voi kirjoittaa. Kuitenkin luokat heittävät vääristä syötteistä/“väärään aikaan” kutsutuista metodeista poikkeuksia.

Vaatimus: V29. Ohjelma on riittävän helppo käytettäväksi Tito-kurssilla

Täyttyikö vaatimus: Kyllä.

Huomautuksia: Ks. V1.

6.4.2 Megaohjelman suoritus

Testin nimi: Mega-ohjelman suoritus.

Testin syötteet: mega.k91-ohjelma, joka sisältää kaikki TTK-91-konekielen käskyt sekä muistiosoitukset (ks. viite 2).

Haluttu tulos: Ohjelma tulee voida suorittaa ilman virheilmoituksia.

Testin tulos: Ok.

Testin nimi: Mega-ohjelman binäärimuodon käännös takaisin symboliseksi.

Testin syötteet: mega.b91-tiedosto, joka saatiin kun mega.k91 käännettiin.

Haluttu tulos: Ladatun mega.b91-ohjelman binääreistä käännetyt symboliset käskyt vastaavat mega.k91:n symbolisia käskyjä.

Haluttu tulos: Ok.

Liite 1. Testaajien lausunnot

Alfa- ja betatestaus

Toteutusryhmän ulkopuolisia testaajia oli kolme. Yksi osallistui alfatestaukseen ja betatestasi hieman tämän jälkeen itsekseen, muttei löytänyt enää muuta huomautettavaa. Toiselle ohjelmaa demottiin lyhyesti, minkä jälkeen hän betatestasi ohjelmaa itsekseen. Kolmas toimi vain betatestaajana. Testaajat olivat ensimmäisen, toisen ja n:nen vuoden opiskelijoita, tässä järjestyksessä. Ensimmäisen vuoden opiskelija oli juuri käynyt Tietokoneen toiminta -kurssin, ja toisen vuoden opiskelija lisäksi harrastanut assembleria, joten täysin ummikkoja emme saaneet testaajiksi. Kaiken kaikkiaan testaajat vaikuttivat tyytyväisiltä kokonaisuuteen, ja kommentoivat lähinnä käyttöliittymään liittyviä asioita. Seuraavassa käsitellään testaajien huomautuksia ja kommenttejamme niihin.

Tiedostonhallintaa

Testaajan huomautus: Viimeksi avatun tiedoston nimi olisi hyvä näkyä jossakin.

Ryhmän kommentti: Ryhmä on samaa mieltä.

Testaajan huomautus: Uudelleenlatausnappi olisi kätevä. (Esimerkiksi käännöksen onnistuttua tai suorituksen päätyttyä saman koodin avaaminen uudelleen.)

Ryhmän kommentti: Ryhmä on jokseenkin samaa mieltä; myös viimeksi avattujen tiedostojen nopea saatavuus voisi olla toivottavaa. Toimintoa tullee kuitenkin kutsua uudelleenavaukseksi, ettei mennä sekaisin automaattisesti tapahtuvan latauksen kanssa.

Testaajan huomautus: Koodin muokkaus lennossa oli mukava lisä; uusien koodirivien luomismahdollisuus olisi parannus tähänkin.

Ryhmän kommentti: Asiakkaan vaatimusten mukaan koodin muokkausta ei tässä vielä toteutettu. Epäilemättä “avaa uudelleen” -nappi ja/tai muokkausmahdollisuuksien laajentaminen olisivat hyviä käytettävyyslisiä ohjelmistoon.

Testaajan huomautus: Jos koodia voisi muokata kunnolla, “uusi” ja “tallenna” -napit olisivat tarpeen.

Ryhmän kommentti: Ryhmä on samaa mieltä. Etenkin tallentamisen muuttaminen automaattisesti tapahtuvasta loogisemmaksi, tallennusnapista tapahtuvaksi, voisi olla edullista. Myös “tallenna nimellä” -nappi on ehdotuslistalla.

Testaajan huomautus: Eikö binääritiedoston sisältö voisi olla oikeaa binääriä?

Ryhmän kommentti: Kun tästä sovittiin asiakkaan kanssa alunperin, tultiin siihen tulokseen että sisältöä tulisi voida lukea omin silmin tavallisen tekstitiedoston muodossa. Tämä rajaa pois joitakin binääritiedoston toteutusvaihtoehtoja.

Rekisterit, symbolitaulu

Testaajan huomautus: Rekisterien R6 ja R7 vieressä pitäisi lukea SP ja FP animaatioikkunassa, ja peruskäyttöliittymässä vastaavasti SP:n ja FP:n vieressä tulisi lukea R6 ja R7. Testaaja totesi myös, että samaan rekisteriin viittaaminen kahdella nimellä on sairasta.

Ryhmän kommentti: Ryhmä on samaa mieltä alusta. Jälkimmäinen, TTK-91:stä lähtöisin oleva konventio kaksinimisistä SP:stä ja FP:stä helpottanee kuitenkin binäärimuodon ymmärtämistä.

Testaajan huomautus: Tilarekisteri puuttuu peruskäyttöliittymästä ja siinä on vain 3 bittiä animointi-ikkunassakin. Missä loput bitit ovat? Pitäisikö niitä tiivistää? ["..." tulkittiin merkiksi bittien mahtumattomuudesta.] Miksei tilarekisteri näy kokonaislukuna muiden rekisterien tapaan? Sinne pitäisi voida halutessaan myös kirjoittaa jonkun tilan.

Ryhmän kommentti: Animaatioikkunassa näkymättömien bittien funktio on hieman epämääräinen. Sisäisesti poikkeusbitit on korvattu Javan poikkeuksilla, jolloin bitin voisi nostaa pystyyn suorituksen päättyessä; bittejä ei kuitenkaan voi tällä hetkellä saada RunInfosta, koska ajonaikaisen poikkeuksen heiton jälkeen sellaista ei ole saatavilla. SVC-kutsua ja käyttöjärjestelmäkoodia ei animoida, koska TTK-91:ssä ja Tietokoneen toiminta -kurssissa käyttöjärjestelmä on piilotettu lähes täysin. Täten SVC-kutsua, käyttöjärjestelmätilaa, laitteelta tulevaa keskeytystä tai keskeytysten pois päältä kytkemistä merkitsevät bitit eivät olisi pystyssä kuin hetken kesken SVC- ja/tai IN/OUT-käskyjä, joista SVC-komentoja ei tällä hetkellä animoida lainkaan. Loput 21 bittiä (tilarekisteri on muiden rekisterien tapaan 32-bittinen) ovat täysin määrittämättä.

Jos tilarekisteri olisi numeroarvo muiden rekisterien tapaan, se kertoisi kieltämättä käyttäjälle selkeästi, ettei statusrekisteri ole sen kummoisempi kuin muutkaan rekisterit. Rekisterin sisältöä olisi kuitenkin kenties jo liian hankala lukea. Ehkä tarvittaisiin vaihtoehtoiset esitysmuodot tällekin?

Tilarekisteriä ei voi suoraan muokata. Onko joku kieli, jossa näin voi tehdä? Tämän voisi kuvitella aiheuttavan melkoisia ongelmia, kuten "siirränpä itseni käyttöjärjestelmätilaan ja estänpä poikkeukset kirjoittamalla sopivan luvun tilarekisteriin"?

Peruskäyttöliittymästä tilarekisteri jäi osin edellä mainitun abstraktio-ongelman takia; sen tietoja ei laitettu lopulta suoraan rekisteriin, vaan tiedot täytyy hakea RunInfoista. Processorissa näkyvä, yhteis-APIinkin päässyt "status" on itse asiassa vain "ollaanko suorituksessa, vai loppuiko se jo, normaalisti vaiko epänormaalisti" -merkintä.

Testaajan huomautus: Symbolitaulusta saa hieman sellaisen kuvan, että se olisi jotenkin koneen sisällä rekistereitä ja muistia vastaavassa asemassa, mikä hämännee käyttäjiä.

Ryhmän kommentti: Symbolitaulu päätettiin pitää näkyvissä koodin seuraamisen helpottamiseksi, mutta oikean “koneen” voisi tosiaan rajata siten, että symbolitaulu ei näytä olevan sen osa.

Testaajan huomautus: Vanhan Koksin tapaan käännös- ja suoritusprosessin eri vaiheet eivät ole täysin selviä, koska kaikki tehdään samassa tilassa.

Ryhmän kommentti: Kenties tähän voisi jatkokehityksessä kiinnittää vielä lisää huomiota. Käännöstilan ja muistin erottaminen kahdeksi omaksi alueekseen voi viedä turhan paljon tilaa, mutta auttaisiko esimerkiksi jonkinlainen värikoodaus?

Yleisiä kommentteja selkeydestä ja käytettävyydestä

Testaajan huomautus: Valittu muistin koko ja kieli olisi hyvä näkyä selkeämmin.

Ryhmän kommentti: Ryhmä on samaa mieltä.

Testaajan huomautus: Näppäimistösyötettä pyydettäessä fokus saisi siirtyä KBD-kenttään suoraan.

Ryhmän kommentti: Ryhmä on samaa mieltä.

Testaajan huomautus: Näppäimistösyötettä pyydettäessä animaatioikkunaan saisi ilmestyä KBD-kenttä myös.

Ryhmän kommentti: Kirjataan loppuraporttiin jatkokehitysehdotukseksi.

Testaajan huomautus: Asetusvalikossa on “set” ja “configure” -lisäsanoja, joita ei välttämättä tarvitse – ollaanhan jo valmiiksi asetusvalikossa.

Ryhmän kommentti: Kirjataan ehdotus loppuraporttiin.

Testaajan huomautus: Normaalisti debuggerit korostavat sen rivin, joka suoritetaan ’next’-nappulan painamisen jälkeen. Titokoneessa korostetaan juuri äsken suoritettu rivi.

Ryhmän kommentti: Seuraamme Koksin esimerkkiä, mutta tämä muutos saattaisi olla kokeilun arvoinen.

Itse TTK-91-konekieli

Testaajan huomautus: XOR on rikki; XOR R0, R0 ei anna tulokseksi nollaa, vaikka minkä tahansa eXclusive OR itsensä kanssa on tulokseltaan 0.

Ryhmän kommentti: Tämä on TTK-91-kielen ominaisuus. Ilmiö johtuu R0:n erikoisroolista toisena parametrinä; vaikka siihen voi tallentaa lukuja, toisena parametrinä käytetty R0-viittaus tulkitaan aina sisällöltään 0:ksi. Tämä siksi, että komennon binäärimuodossa ei voida sanoa “tässä komennossa ei ole indeksirekisteriä”, jolloin esimerkiksi JUMP 3 on sama kuin JUMP 3(R0). Koska ensimmäinen parametri on aina vain yksi rekisteri (joka tosin voidaan jättää merkitsemättä jos sitä ei käytetä mihinkään, kuten JUMP-käskyssä), siellä ongelmaa ei ole.

Testaajan huomautus: STORE on rikki; STORE R1, =3 ei mene kääntäjästä läpi.

Ryhmän kommentti: Tämä on TTK-91-kielen ominaisuus, joka on päätelty Koksin käytöksestä. Ilmiö johtuu siitä, että STOREsta vähennetään aina yksi muistinouto (viimeinen muistissa käynti johtuu itse tallennuksesta), jolloin STORE R1, 3 tallentaa R1:n sisällön muistiriville 3 ja STORE R1, @3 tallentaa R1:n sisällön muistirivillle, jonka osoite on tallennettu muistiriville 3. Tällöin STORE R1, =3 vähentäisi muistinoutojen määrän -1:een, eikä sillä ole merkitystä. Toisaalta, STORE tallentaa annetussa rekisterissä olevan tiedon muistiin. Jos tämä tieto tallennettaisiin kokonaislukuun, sitä käsiteltäisiin muuttujana mikä ei ole mahdollista missään oikeassa kielessä tai matematiikassa.

Toinen muistiosoitteita käyttävä komentoryhmä ovat hyppykäskyt. JUMP =3 kääntyy Koksissa muotoon 32:6:0:0:0 (2. operandi puuttuu täysin), kun taas JUMP 3 kääntyy muotoon 32:6:0:0:3. CALL ei hyväksy muistinosoitusmoodiksi ’=’:a lainkaan. Titokoneessa JUMP ei käänny vakioparametrillä, ja CALLissa molemmat muodot (välitön operandi ’=’:lla ja suora muistiosoitus) kääntyvät samalla tavalla (=3 ja 3 -> 3).

Muistin esitys

Testaajan huomautus: Kaksoispisteillä erotettu käskyjen esitys (esim. NOP R3, =1(R2) -> 0:3:0:3:1) koodista olisi hyödyllisempi kuin kokonaislukuesitys; toisaalta desimaali puolustaa paikkaansa muistiin tallennetuissa kokonaislukuarvoissa.

Ryhmän kommentti: Edellinen huomio esitettiin myös asiakkaalle demotilaisuudessa. Tämä ehdotti, että jatkoversiossa voisi olla valintamahdollisuus sille, minkämuotoinen merkistö näytetään. Kenties oma valinta data- ja koodialueille olisi tarpeen.

Testaajan huomautus: Symboliset viittaukset häviävät koodin seasta jos se muokkaa itseään. Kun rivin “X MOV R0, =5” päälle kirjoitetaan komennon “MOV R0, =4” binäärikoodi, koodissa pitäisi lukea “X MOV R0, =4”, koska X ei ole sidonnainen koodisisältöön. Näin ei kuitenkaan käy. Jos käyttäjää “huijataan” symbolisella muodolla, josta näkyy lisätietoja, huijaus tulisi tehdä loppuun asti...

Ryhmän kommentti: Ryhmä valitsi nykyisen, binääristä päättelyä informatiivisemman muodon symboliseen esitykseen, koska se seuraa Koksin koodinäytön periaatetta. Kommentit ja symbolit jätettiin mukaan, koska ne helpottanevat koodin lukua ja

sitä, että käyttäjä tunnistaa käännetyn binäärikoodin toiseksi, vastaavaksi versioksi omasta symbolikielisestä koodistaan. Tämä kuitenkin tosiaan aiheuttaa tiettyä epäloogisuutta, kun koneella ei olekaan oikeasti käytössään muistiviite- ym. ylimääräistä tietoa. Binääriä purettaessa käytetään toisaalta oikeaa päättelyä, toisin kuin esimerkiksi Koksin “numeerisessa tallennuksessa” / muistidumpissa.

Testaajan huomautus: Symboliset viittaukset ym. voisi erottaa itse koodista vaikka harmaalla, koska ne eivät ole olemassa. Lisäksi taustavär(e)illä voisi merkitä nimettyjä kohtia kuten X yllä.

Ryhmän kommentti: Värien vaihto kesken sanan ei vaikuttanut kovin yksinkertaiselta. Kovin villi värimaailma ei lienisi sekään eduksi.

Testaajan huomautus: Symbolista muotoa ei ehkä pitäisi näyttää automaattisesti ainakaan binääriä ladattaessa, vaan jostain pitäisi painaa “disassemble”-nappia, jolloin “päätellyt” arvot tulisivat vasta näkyviin.

Ryhmän kommentti: Ryhmän maku ei aivan täsmää, mutta ehdotus kirjataan loppuraporttiin.

Testaajan huomautus: Sarakkeet ovat normaalikäyttöä varten oudossa järjestyksessä; länsimaissa kun tehdään jollekin jotain, tulee uusi versio oikealle puolelle. Eli symbolisen version tulisi olla vasemmalla ja binäärin oikealla. Vastaavasti kuitenkin binääriä ladattaessa “disassembloidun” symbolisen koodin tulisi olla oikealla puolella. Jos koodista käännetty softa vielä erikseen disassembloitaisiin, sarakkeiden tulisi olla järjestyksessä alkuperäinen koodi – binääri – disassembloitu koodi. Ja mukaan vielä assemble/disassemble-napit. [Tästä on piirretty esimerkki, kuvana 1.]

Ryhmän kommentti: Sarakkeiden järjestyksen automaattinen vaihtaminen saattaisi olla vieläkin hämmentävämpää.

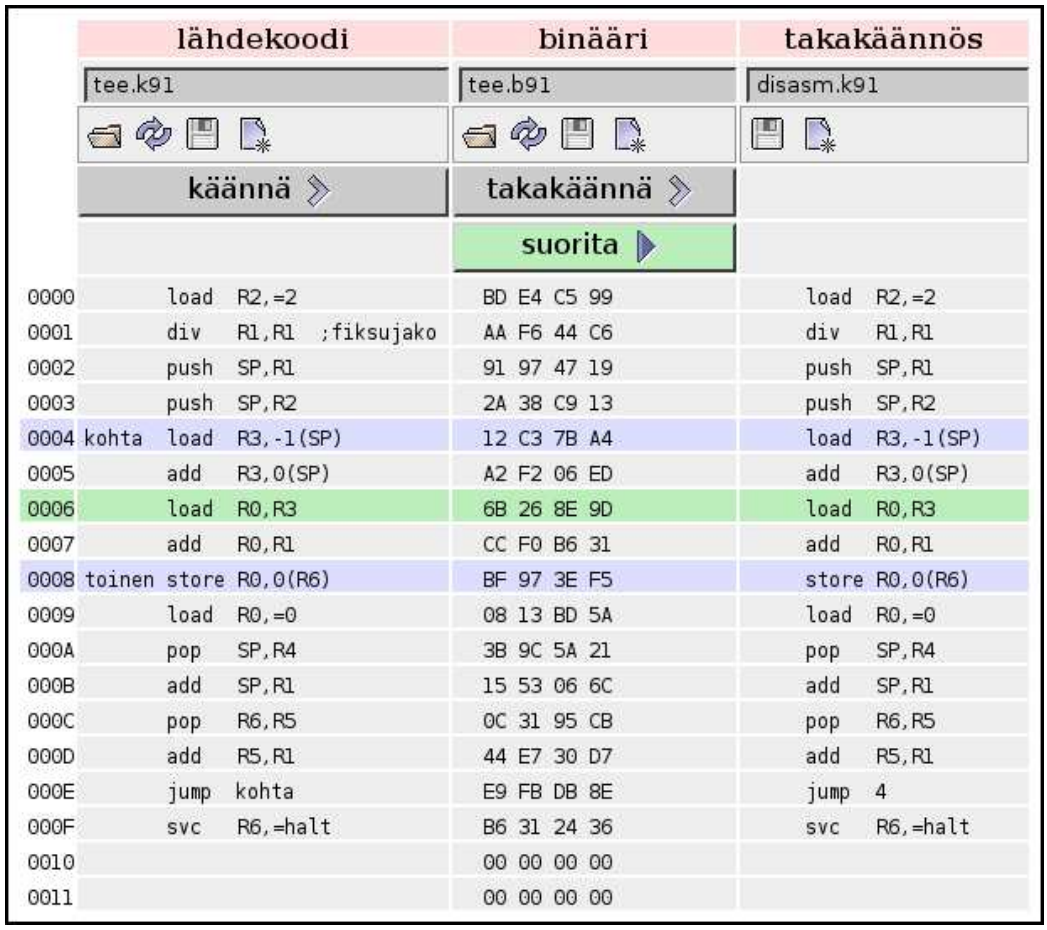
Sekalaisia havaintoja

Testaajan huomautus: Kerolan mystery-esimerkkiohjelma (itseään muuttavaa koodia) ei ole enää mysteeri, koska vastauksen näkee jo suoraan!

Ryhmän kommentti: Ilmeisesti kuitenkin se, että tulosta ei välttämättä näe suoraan koodista, jättää riittävästi mysteeriä.

Testaajan huomautus: “Kääntäjä” on liioiteltu termi, tässä kuitenkin tehdään 1:1-muunnos symbolisesta konekielestä binääriksi. Tosin ehkä ohjelmassa on kuitenkin viisaampaa käyttää kurssin termistöä.

Ryhmän kommentti: Ryhmä suosii kurssin termistöä. Jos suomen kielestä löytyisi hyvä käännös “assemblerille”, tätä voisi myös ehdottaa käytettäväksi.



		lähdekoodi		binääri	takakäännös
0000		load R2,=2		BD E4 C5 99	load R2,=2
0001		div R1,R1 ;fiksujako		AA F6 44 C6	div R1,R1
0002		push SP,R1		91 97 47 19	push SP,R1
0003		push SP,R2		2A 38 C9 13	push SP,R2
0004	kohta	load R3,-1(SP)		12 C3 7B A4	load R3,-1(SP)
0005		add R3,0(SP)		A2 F2 06 ED	add R3,0(SP)
0006		load R0,R3		6B 26 8E 9D	load R0,R3
0007		add R0,R1		CC F0 B6 31	add R0,R1
0008	toinen	store R0,0(R6)		BF 97 3E F5	store R0,0(R6)
0009		load R0,=0		08 13 BD 5A	load R0,=0
000A		pop SP,R4		3B 9C 5A 21	pop SP,R4
000B		add SP,R1		15 53 06 6C	add SP,R1
000C		pop R6,R5		0C 31 95 CB	pop R6,R5
000D		add R5,R1		44 E7 30 D7	add R5,R1
000E		jump kohta		E9 FB DB 8E	jump 4
000F		svc R6,=halt		B6 31 24 36	svc R6,=halt
0010				00 00 00 00	
0011				00 00 00 00	

Kuva 1: Betatestaajan piirtämä esimerkki sarakkeiden järjestyksestä, käytetty luvalla. Siniset rivit ovat symbolein merkittyjä kohtia, vihreä rivi esittää suorituksen etenemistä.

Testaajan huomautus: Sisennykset ovat väärin joissakin luokissa, ja jotkin rivit ovat yli 80 merkkiä. (Viittaus Java Coding Conventionsiin.)

Ryhmän kommentti: Ryhmä ei tiennyt etukäteen kyseisestä ohjeistuksesta, eikä projektin loppupuolella päätetty enää lähteä jälkijättöisesti korjaamaan jälkeä. Suurin osa luokista on konsistentisti sisennetty, mutta ilmeisesti jotkin tekstieditoreistamme eivät ole aivan yhteensopivia.

Liite 2. mega.k91

```
;KÄSKYT
MUUTTUJA DC 1000


TOINEN  DS  4
TUHAT   EQU 1000
STDIN   DEF TIEDOSTO
STDOUT  DEF TIEDOSTO

   NOP

   LOAD  R1, =100
   STORE R1, MUUTTUJA
   IN    R1, =KBD;ennen lopputestejä
   OUT   R1, =CRT

   ADD  R1, =1
   SUB  R1, =1
   MUL  R1, =1
   DIV  R1, =1
   MOD  R1, =1
   AND  R1, =1
   OR   R1, =1
   XOR  R1, =1
   SHL  R1, =1
   SHR  R1, =1
   SHRA R1, =1

   COMP R1, =1

   JUMP A1
A1 JNEG  R1, A2
A2 JZER  R1, A3
A3 JPOS  R1, A4
A4 JNNEG R1, A5
A5 JNZER R1, A6
A6 JNPOS R1, A7

A7 JLES  R1, A8
A8 JEQU  R1, A9
A9 JGRE  R1, B1
B1 JNLES R1, B2
B2 JNEQU R1, B3
```

```
B3 JNGRE R1, B4
B4 NOP

   CALL  SP, AO
   PUSH  SP, =1
   POP   SP, R1
   PUSHR SP
   POPR  SP  ;SVC LOPUSSA

;REKISTERIT

   STORE SP, MUUTTUJA

   LOAD  R0, =99
   LOAD  R1, =100
   LOAD  R2, =101
   LOAD  R3, =102
   LOAD  R4, =103
   LOAD  R5, =104
   LOAD  R6, =105
   LOAD  R7, =106
   LOAD  R0, R1
   LOAD  R1, R2
   LOAD  R2, R3
   LOAD  R3, R4
   LOAD  R4, R5
   LOAD  R5, R6
   LOAD  R6, R7
   LOAD  R7, SP

   LOAD  R0, MUUTTUJA
   LOAD  R1, MUUTTUJA
   LOAD  R2, MUUTTUJA
   LOAD  R3, MUUTTUJA
   LOAD  R4, MUUTTUJA
   LOAD  R5, MUUTTUJA

   LOAD  R2, =100

   LOAD  R1, TUHAT
   LOAD  R1, =100(R2)
   LOAD  R1, MUUTTUJA(R2)
   LOAD  R1, @MUUTTUJA
   STORE R1, TOINEN
   LOAD  R0, =1
```

```
   LOAD  R1, =1
   STORE R1, TOINEN(R1)  ;HÄMÄYS. JOS R1 ON SEURAAVAN
                         ;JÄLKEEN 1 NIIN VÄÄRIN.
   LOAD  R1, @TOINEN     ;PITÄISI OLLA SAMA KUIN MUUTTUJASSA
   LOAD  R1, @TOINEN(R0) ;R0 TOISESSA PARAMETRISSÄ ON AINA NOLLA
   SVC   SP, =HALT

AO EXIT SP, =0
```