TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond
Arvutisüsteemide instituut

IA40LT
Marek Grencštein 142480IASB

SKRIPTIKEELTE KASUTAMIINE REAALAJAS ARENDAMISEKS

Bakalaureusetöö

Juhendaja: Vladimir Viies
PhD

Tallinn 2017

Autorideklaratsioon

Kinnitan, et olen koostanud antud lõputöö iseseisvalt ning seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teiste autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on töös viidatud.

Autor: Marek Grencštein

22.05.2017

BAKALAUREUSETÖÖ ÜLESANNE

Üliõpilane: Marek Grencštein

Üliõpilaskood: 142480IASB

Lõputöö teema eesti keeles: Skriptikeelte kasutamine reaalajas arendamiseks

Lõputöö teema inglise keeles: Scripting languages for real time devlopment

Juhendaja (nimi, töökoht, teaduslik kraad, allkiri): Dots. Vladimir Viies, Tallinna Tehnikaülikool, Arvutisüsteemide instituut, PhD.

Konsultandid: Hannes Kinks

Lahendatavad küsimused ning lähtetingimused: Reaalajas arendustöödeks mõeldud skriptikeele valmistamine eeldusel, et tuleb koostada oma keel.

Eritingimused:

Nõuded vormistamisele: Vastavalt Infotehnoloogia teaduskonnas kehtivatele nõuetele

Lõputöö esitamise tähtaeg: 22.05.2017

Ülesande vastu võtnud: _____ kuupäeva:

(lõpetaja allkiri)

Annotatsioon

Arendustegevuses on oluline kasutada arendusaega efektiivselt. Suuremate süsteemide puhul läheb palju aega koodi kompileerimiseks ja süsteemi viimiseks testitavasse olekusse. Samuti on paljud sellised suuremad süsteemid kirjutatud programmeerimise keeles, milles ei ole võimalik kindla ülesande jaoks kiiresti arenduskoodi kirjutada. Selliseid probleeme saab lahendada teostades reaalajas arendamist kasutades skriptikeeli.

Bakalaureusetöö põhieesmärgiks on tutvustada skriptikeelte kasutamist reaalajas arendustegevuses ja kiirendada arendusprotsessi uue skriptikeelse lahendusega.

Töö tulemusena valmis reaalajas arendustööks sobiv tööriist – SCR –, mis võimaldab programmeerijal teostada reaalajas arendamist ülesande jaoks sobiva süntaksiga skriptikeeles ja kiirendada testimiseks kuluvat aega säilitades suures ulatuses programmis esinenud muutujate väärtused kahe järjestiku töödeldud skriptiversiooni vahel.

Lõputöö on kirjutatud eesti keeles ning sisaldab teksti 22 leheküljel, 3 peatükki, 10 joonist.

Abstract

Skripting languages for real time development

Software development process can be time consuming. Writing programming code to fit the system requirements takes a lot of time. In addition, every time new code gets added to the program, it needs to be compiled down to machine code. Even for small programs this takes noticable time that scales as the software grows. Moreover, starting program and taking it to the state where it can be tested takes additional time. This time is spent not doing productive development work and leads to projects that take longer to complete.

Using scripting languages to perform realtime development saves development time due it removes the need to re-compile and in some cases program doesn't even need to be restarted after the code has changed. These scripting languages will require API to integrate it into the system.

As a result of the thesis, an API called SCR was created to develop system writen in C/C++ programming language. SCR places it's main focus on restoring memory data between executions of two successive script versions. As a default, resulting scripting language supports writing scripts only in assembly-like syntax. It is possible to define custom syntax filter to be used while loading script to support more task-oriented development. This leads to increase in development speed.

In conclusion, there are already many scripting languages that can be used in realtime development. SCR takes one step further to increase development speed via keeping script memory/state throughout functionality development. It is required to be cautious about the validity of the data stored in script memory and without writing custom filters script writing in assembly can cause slow-down in development for more complex functionality.

The thesis is in estonian and contains 22 pages of text, 3 chapters, 10 figures.

Lühendite ja mõistete sõnastik

API	*Application Programming Interface* – rakenduste programmeerimises kasutatav üldise kasutamisliidesega teek; pakub funktsionaalsust, mida programmist saab oma konkreetse programmi kirjutamisel kasutada.
assembler	Antud töös: inimesele loetavalt kirjutatud riistvarale lähedane programm, mille igale käsule saab panna üks-ühele vastavusse käsukoodi.
bait	Arvuti mälus väikseim adresseeritav ühik. Mälu suuruse ühik. Tavaliselt sisaldab 8 bitti ehk binaarset numbrikohta.
char	Tavaliselt ühebaidine tähemärgina esitatav andmetüüp.
double	Tavaliselt 8 baidine topelt täpsusega ujupunktarvuna esitatav andmetüüp.
int	Tavaliselt 4 baidine täisarvuna esitatav andmetüüp.
mnemoonika	Sümbolitest koosnev muster, mis aitab midagi mäletada.
string	Arvutiprogrammis tähtede massiiv. Tekst.
süntaks	Süntaks on grammatika osa, mis käsitleb sõnade ühendamist sõnaühenditeks ja lauseteks.
viit	Programmeerimise kontekstis – mäluaadress. Viit osutab mälus asuvatele andmetele. Mõnel juhul viitab koodis asuvale funktsioonile.

Sisukord

Jooniste loetelu

Sissejuhatus

Arendustegevuses on oluline kasutada arendusaega efektiivselt. Suuremate süsteemide puhul läheb palju aega koodi kompileerimiseks ja süsteemi viimiseks testitavasse olekusse. Samuti on paljud sellised suuremad süsteemid kirjutatud programmeerimise keeles, milles ei ole võimalik kindla ülesande jaoks kiiresti arenduskoodi kirjutada. Selliseid probleeme on võimalik lahendada kasutades skriptikeeli ja teostada seeläbi reaalajas arendustegevust.

Antud bakalaureusetöö põhieesmärgiks on tutvustada skriptikeelte kasutamist reaalajas arendustegevuses ja kiirendada arendusprotsessi uue skriptikeelse lahendusega. Töö on jaotatud kolmeks osaks.

Esimeses osas tutvustatakse skriptikeele olemust ja reaalajas arendamiseks sobivaid skriptikeeli, tutvustades täpsemalt kahte populaarset skriptikeelt – Lua ja Python.

Teises osas tutvustatakse üldiseid põhimõtteid skriptikeelte kasutamisel reaalajas arendamiseks. Lua ja Python'i näitel antakse ülevaade nende ühendamiseks programmeerimise keelega C.

Töö kolmandas osas kirjeldatakse autori poolt loodud C/C++ keeles kasutatav reaalajas arendamise tööriista – SCR. Välja on toodud selle üldised omadused, ülesehitus, tööpõhimõte ja reaalajas arendustegevuseks kasutamine.

1 Skriptikeeled

Skriptikeel on kõrgetasemelisem programmeerimise keel, kus programmi käske mõistetakse ja täidetakse ühekaupa [1].

Maailmas leidub väga palju erinevaid skriptikeeli. Suurem osa neist sobivad laiaotstarbeliseks kasutuseks. See tähendab, et neid sobib kasutada igasuguste ülesannete lahendamiseks ja need on teistest programmeerimise keeltest sõltumatud. Lisaks leidub ka probleemile orienteeritud skriptikeeli, mida kasutatakse kindlate probleemide lahendamiseks.

1.1 Mis on skriptikeel

Üldiselt on skriptikeeltes programmeerimine lihtsam ja vähemnõudlikum. Seetõttu toimub nendes programmeerimine/arendamine kiiremini kui rohkem struktureeritud kompileeritavates keeltes, näiteks C või C++ [1].

Erinevalt kompileeritavatest keeltest, mis tõlgitakse kompilaatori poolt masinkoodi enne, kui programmi saab arvutis tööle panna, muudetakse skriptikeeles kirjutatud programm arvuti poolt arusaadavaks alles selle töötlemisel interpretaatori poolt. Selline lähenemine on aeglasem kui kompileeritavatel keeltel, sest käsu täitmine ei ole enam täielikult protsessorist sõltuv vaid vajab eraldi ühe käsu haaval programmi – interpretaatori – poolt töötlemist [1].

Skriptikeelte edasiarenemisega on skriptikeelteks kategoriseerimine hägustunud. Üheks põhjuseks on skriptikeelte töötlemise viiside täiustumine, kus mõnel juhul toimub enne programmi täitmist masinkoodi kompileerimine (*just-in-time compilation*). Tänapäeval ei ole enam võimalik nii kindlalt piiritleda, mida skriptikeel peaks tegema ja kuidas käituma. Vaikimisi on jõutud kokkuleppele, et kõrgkeele klassifikatsiooni ei määra enam keel ise, vaid see, kuidas keelt kasutatakse [1]. Skriptikeelte kasutusalad võib jagada nelja kategooriasse: käsuskriptikeeled (*command scripting languages*), rakendus-skriptikeeled

(*application scripting languages*), märkekeeled (*markup languages*) ja universaalsed skriptikeeled (*universal scripting languages*) [2].

Tihti kasutatakse skriptikeeli ka erinevate süsteemikomponentide ühildamiseks. Sellises kontekstis kutsutakse neid ka liimikeelteks *(glue language)* [1]. Liimikeel ei paku suuremale süsteemile mingit täiendavat funktsionaalsust, kuid võimaldab ühendada erinevaid süsteemiprotsesse ja väiksemaid komponente omavahel, mis ilma teineteisega suhelda ei saaks. Sellist lahendust kasutatakse tihti kiirete prototüüpide valmistamisel [3], [1].

1.2 Reaalajas arenduseks sobivad skriptikeeled

Kõige mahukama skriptikeelte klassi moodustavad nö laiaotstarbelised keeled. Need keeled on oma disainilt sobilikud igasuguste ülesannete lahendamiseks. Lisaks laiaotstarbelistele skriptikeeltele, mis on täiesti iseseisvad, leidub ka keeli, mille disainis on ette nähtud kindel kasutusala. Tavaliselt on need keeled sobilikud mingi konkreetse probleemi lahendamiseks. Näiteks rakenduse jaoks, mille arendamiseks on vaja teha palju maatriksarvutusi, sobib keel *matlab*, mille süntaks on disainitud selle eesmärgiga.

Programmeerimiskeele liigituse skriptikeeleks määrab pigem see, kas seda kasutatakse skriptikeelena või mitte. Seetõttu on kasutusvalik lai. Julgelt võib nimetada skriptikeelena ka selliseid keeli nagu *Java* ja *Lisp*, mis traditsiooniliselt ei arvestata skriptikeelte hulka. Lisaks kuulub skriptikeelte alla palju teisi keeli nagu *Perl*, *PHP* ja *Javascript*. Antud punktis kirjeldatud kaks keelt - *Lua* ja *Python* – on valitud reaalajas arendamiseks suure populaarsuse tõttu.

1.2.1 Lua

Lua on võimas ja kiire programmeerimise keel, mida on lihtne õppida, kasutada ja liita rakendustega [4]. See on dünaamiliselt kirjutatav programmeerimise keel, mille baidikoodi interpreteeritakse registripõhisel virtuaalmasinal [4]. Lua on ennekõike lihtsalt teek, mida saab liita teiste rakendustega sisaldab endas ainult hädavajaliku funktsionaalsust [5].

Lua programm koosneb märkidest (*token*), milleks on võtmesõna, identifikaator, konstant, sõna (*string literal*) või sümbol. Koodi kirjutamiseks on 21 võtmesõna: *and*,

break, do, else, elseif, end, false, for, function, if, in, local, nil, not, or, repeat, return, then, true, until ja *while*, mis on reserveeritud ja ei saa kasutada konstantide või muutujate nimedena. Traditsiooniliselt peavad muutujanimed algama kas alakriipsu või tähega ja nimed on tähesuurusetundlikud, mille tõttu nähakse näiteks deklareeritud muutujanimesid *var* ja *Var* erinevate muutujatena [6], [7]. Kuigi esialgu näeb Lua kood eemalt vaadates välja väga sarnane Pythonis kirjutatud koodile - koodis puuduvad semikoolonid, loogelised sulud koodiplokkide eraldamiseks ja koodiplokkide joondamise reeglid näivad sarnased - on Lua's koodi vormindamine vaba ja vormindamise reegleid peaks ennekõike järgima koodi loetavuse pärast. Koodis lausete eraldamine käib tühikute, reavahetuste ja taanete kasutamisega ja märkide lausesse grupeerimine on interpretaatori töö (vaata näidet süntaksi kohta Joonis 1) [6], [8]. Kui kirjutatakse mitu lauset koodis ühele reale, tuleb kasutada lausete eraldamiseks siiski semikoolonit [9]. Süntaksi näitena on kasutatud väikest programmi, mis arvutab faktoriaali (Joonis 1).

```
--- faktoriaali arvutamine
function fact (n)
    if n == 0 then
        return 1
    else
        return n * fact(n-1)
    end
end

print("Sisesta number:")
a = io.read("*number") -- loe number
res = fact(a)
print(res) -- prindib tulemuse
```

Joonis 1. Faktoriaali arvutamine programmeerimiskeeles Lua

Mitmete kriteeriumite ja testide põhjal loetakse Lua't kiireimaks interpreteeritavaks keeleks. Teiste skriptikeelte kohta kasutatav väljend „sama kiire kui Lua" (*as fast as Lua*) näitab veel enam, et Lua't peetakse väga kiireks interpreteeritavaks skriptikeeleks. Kui interpreteeritavast töötluskiirusest jääb väheks, leidub lisaks Lua tavalisele interpretaatorile ka LuaJIT, mis tõlgib interpreteeritava koodi masinkeelde vahetult enne koodi töötlust [4].

Lua leiab massiliselt skriptikeelena kasutust arvutimängude arenduses, kus kriitilisel kohal on lõpptulemuse kiirus ja mänguloogika arendamisprotsessi paindlikkus [2], [6], [10].

1.2.2 Python

Python on objektorienteeritud kõrgema taseme programmeerimiskeel. Tegu on interpreteeritava dünaamilise skriptikeelega. See sobib kiireks arenduseks ja prototüüpimiseks tänu kõrgetasemelistele sisse-ehitatud andmestruktuuridele ning dünaamilisele kirjutamisele ja komponentide ühildamisele [2], [11].

Pythoni süntaks on lihtne ja paneb suurt rõhku kirjutatud koodi arusaadavusele. See kiirendab oluliselt koodi hooldamisele kuluvat aega [11]. Oma disaini poolest sarnaneb Python C ja Modula segule. Peamine erinevus seisneb koodiplokkide eraldamises taandridadega traditsiooniliste loogeliste sulgude või võtmesõnade asemel [2]. Kõik sama taandega järjestikused read kuuluvad samale koodiplokile ja muutujad on antud ploki ja tema sisse jäävate teiste alamkoodiplokkide suhtes nähtavad. Täpsemalt määrab ära koodis muutuja nähtavuse LEGB (*local, enclosing-functional local, global, built-in*) reegel, mis liigitab muutujad nende deklareerimise järgi kohalikeks , funktsioonisiseste funktsioonide kohalikeks , globaalseteks ja Python'i sisesteks muutujateks [12]. Süntaksi näitena on toodud suvaline Python'is kirjutatud programm (Joonis 2).

```
def add5(x):
    return x + 5

def dotwrite(ast):
    nodename = getNodename()
    label = symbol.sym_name.get(int(ast[0]), ast[1])
    print ' %s [label="%s' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s"];' % ast[1]
        else:
            print '"];'
    else:
        print '"];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print ' %s -> {' % nodename,
        for name in children:
            print '%s' % name,
```

Joonis 2. Pythoni süntaks

Andmetüüpidest on esindatud kõik klassikalised andmetüübid, näiteks täisarvud ja ujupunktarvud. Üldiselt võib andmetüübid jaotada nelja klassi: numbrilised tüübid (*numeric types*), järjestused (*sequences*), kogumid (*sets*) ja kaardid (*mappings*). Järjestuste alla kuuluvad näiteks *string* ja *list* andmetüübid [12]. Kogumite all on Python'is *set*, mis kujutab endast sorteerimata unikaalsete objektide kogumit [13]. Kaardi klassi kuulub assotsiatiivse massiivina töötav sõnastik (*dictionary*) [12]. Python otsustab, milline andmetüüp sobib muutujale vastavalt sellele omistatud väärtuse tüübile töötluse käigus. See tähendab, et samanimeline muutuja võib omada erinevates programmi osades erinevat andmetüüpi väärtusi [2], [11], [12]. Ühelt poolt on see programmeerija jaoks mugav, kuid teiselt poolt avab koodi vigadele.

Pythoni interpretaatorid on saadaval mitmetele operatsioonisüsteemidele, mistõttu leiab see kasutust mitmes valdkonnas. Alates aastast 2003 on Python olnud kümne populaarseima programmeerimiskeele seas. Veebirakenduste arendus on üheks populaarseimaks valdkonnaks, kus antud skriptikeelt kasutatakse. Lisaks kasutatakse Pythonit ka koos kompileeritava keelega, et kiirendada arendusprotsessi või ettevõttesiseste arendustööriistade kirjutamiseks (näiteks testimistööriistad). Väga edukalt on integreeritud Python'it ka suuremate tarkvaraliste toodete skriptikeelena, näiteks 3ds Max, Blender, Maya, Softimage, Nuke, Gimp jt [12], [14].

2 Skriptikeelte kasutamine reaalajas arendamiseks

Skriptikeeli kasutatakse tihti koos kompileeritavate keeltega. Isegi väiksematel süsteemidel on kompileerimisele kuluv aeg suur, kui kompileerimist tuleb läbi viia väga tihti. Selle aja jooksul ei saa toimuda arendustegevust. Peale igat väikest muudatust uuesti programmikoodi kompileerimine, käivitamine ja muudatuse testimine võtab kokkuvõttes palju aega, seetõttu on otstarbekas rakendada reaalajas arendamist, kasutades skriptikeeli.

2.1 Reaalajas arendamine

Reaalajas arendamiseks on oluline aeg, mis jääb programmi kirjutamise ja kirjutatud tulemuse vahele. Eesmärk on minimeerida seda aega. Vahepealne aeg moodustub koodi uuesti kompileerimisest, programmi käivitamisest ja testitavasse olekusse viimisest.

Kompileerimisele kuluvat aega on võimalik suures osas ära kaotada. Selleks tuleb ühendada skriptikeel abstraktselt tugevas keeles kirjutatud süsteemiga. Alles jääb skriptikeele laadimise aeg.

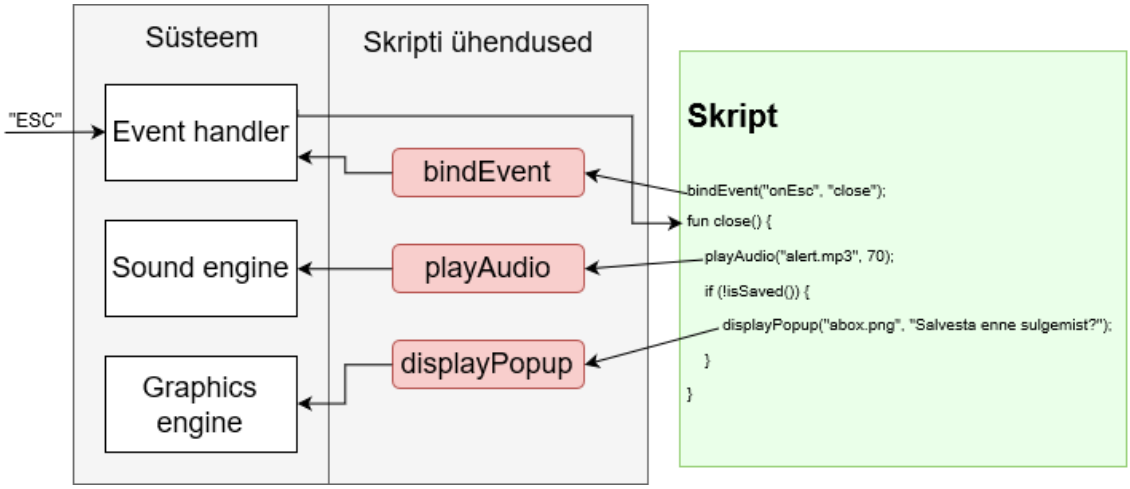
Käivitamisele kuluva aja saab ära kaotada, kui programmi ei pea peale igat muudatust uuesti käivitama. Seega on oluline reaalajas arendamise jaoks, et skripti on võimalik uuesti laadida ilma põhiprogrammi vahepeal sulgemata.

Aeg, mis kulub programmi testitavasse olekusse viimisele võib olla suuremate süsteemide puhul kõige aeganõudvam. Halvimal juhul on vaja selleks sisestada hulk andmeid, anda erinevaid sisendsignaale, laadida faile jne. See kõik võtab aega juba selle tõttu, et seda ei saa või ei ole otstarbekas automatiseerida. Seega on oluline, et reaalajas arendamiseks mõeldud skriptikeel suudab hoida alles eelnevalt kasutuses oleva skriptikeele olekut maksimaalses ulatuses.

2.2 Kasutamismustrid

Kõige klassikalisem arendusmuster skriptide kasutamisel on skriptikoodi ühendamine süsteemi sündmustega. Kompileeritavale programmikoodile lisaks kirjutatakse skripti ühendamise kiht, mis võimaldab skriptile erinevaid teenuseid süsteemi endaga suhtlemiseks. Süsteemisisese sündmuse kuulamine ja sellele reageerimine on juba skripti

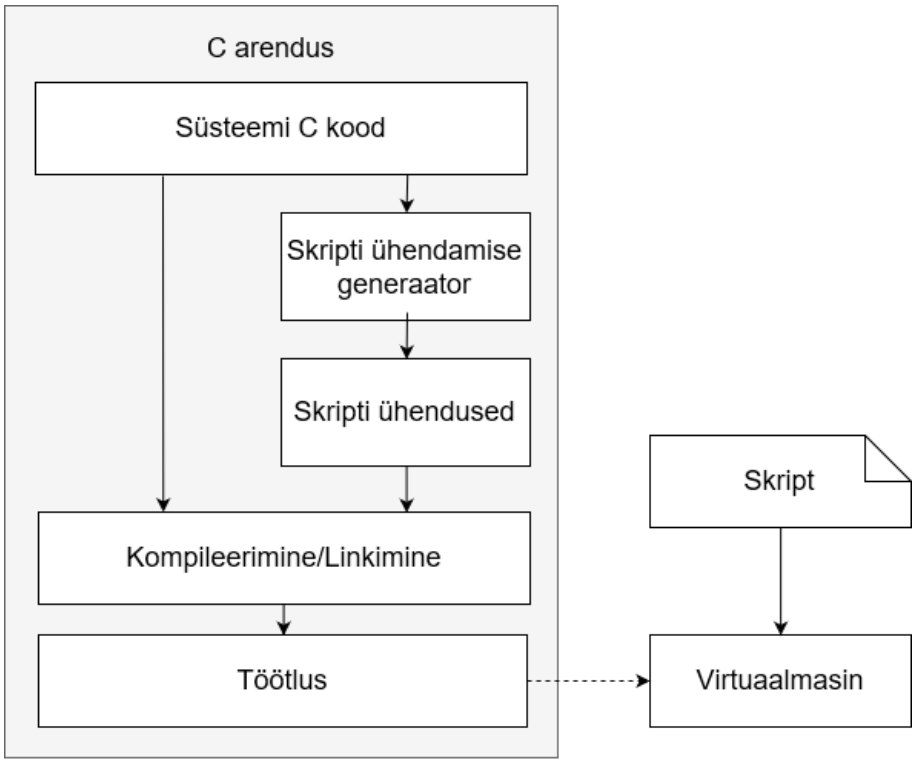
enda teha ja kompileeritavas koodis ei ole vaja teada, kas mõnda sündmust üldse hakatakse kuulama (vaata: Joonis 3) [9], [15].



Joonis 3. Süsteemi sündmuste kasutamine skripti poolt. Näitlik mudel.

Selleks, et skriptikeelt sobiks kasutada reaalajas arendamiseks on vaja tagada kahesuunaline suhtlus skriptikeele ja kompileeritava keele vahel. Kahesuunalisel suhtlusel kasutatakse funktsioonide kohta mõisteid *eksporditavad* ja *imporditavad* funktsioonid. Funktsiooni ekspordiks nimetatakse olukorda, kus kompileeritava keele funktsioon tehakse skripti poolt nähtavaks/kutsutavaks. Import on vastupidine suhtlus, kus skriptikeeles defineeritud funktsiooni kutsutakse kompileeritava keele poolt [9].

Reaalajas arendamise juures kasutatakse põhimõtet, mille järgi kirjutatakse kompileeritavas keeles üks kord, lisades koodis vajaminevatesse kohtadesse ühendusi skriptiga (*hooks*). Üldiselt valmistatakse terve kompileeritav kood suuremal määral abstraktselt, eesmärgiga viia juhtimisega seonduv loogika skripti kätte. Joonis 4 kirjeldab ühendamise üldmudelit, kus skripti jaoks luuakse seosed enne kompileerimist ja linkimist nii, et töötluse ajal oleks võimalik skriptis arendust teostada ilma C koodi muutmata (Joonis 6) [9], [15].



Joonis 4. C koodi ja skripti ühendamise üldmudel

2.3 Skriptide ühendamine

Skriptikeelte ühendamiseks kompileeritava keelega kasutatakse neile vastavat liidest – API, mis tagab skriptide laadimise ning funktsioonide ekspordi ja impordi. Antud alapunktis kirjeldatakse Python'i ja Lua kasutamist C programmeerimiskeelega ja antakse kokkuvõtlik ülevaade vastavalt Python/C ja Lua API'st, mis järgivad sarnast kasutamise mudelit.

2.3.1 Lua

Lua kasutamine koos C (ja C++) programmeerimise keelega käib läbi Lua C API. Selle kasutamiseks on vaja teada, kuidas API's laadida ja töödelda Lua skripti ning teostada funktsioonide eksporti ja importi [9].

Skripti laadimiseks ja töötluseks kasutatakse ühteainsat Lua API funktsiooni nimega *lua_dofile,* mis võtab sisse parameetritena viida Lua olekuobjektile (*lua_State*) ja skriptifaili nimetuse, kusjuures skriptifailina sobib nii varasemalt kompileeritud kui ka kompileerimata skriptikood. Kompileerimata skriptikoodi peamine puudus on väike

ajakulu, mis kulub töötluse käigus kompileerimiseks. Kindlasti ei saa mainimata jätta ka tõsiasja, et koodis võib esineda vigu, mis varasemalt kompileeritud koodi puhul on juba enne programmitöötlust teada ja parandatud (kompilaator ei lase vigast koodi läbi). Need siiski on pisiasjad, kui vaadata Lua kasutamist reaalajas arendustegevuseks [9].

Lua töötab olekute printsiibil (*lua states*). Lua olek sisaldab endas ühe skripti kohta käivat informatsiooni selle töötluskeskkonna kohta. Kui soovitakse kasutada mitut skripti korraga, peab kasutama ka mitut olekut. Lua olekust võib kujutada ette kui virtuaalmasinat, sest see hoiab endas C keele ja Lua omavahelisi ühendusi. Lisaks hoitakse veel kogu andmevahetuseks vajalikku pinu struktuuri (*lua stack structure*) [9].

Funktsiooni eksportimiseks on vaja see pakkida registreerimiseks sobivasse vormi. Funktsiooni prototüüp on kujul *int lua_CFunction(lua_State*).* Sellisel kujul olev funktsioon tehakse Lua skripti jaoks nähtavaks, kasutades *lua_register(lua_State*, const char*, lua_Cfunction)* API funktsiooni, kus vastavalt esimese parameetriks on lua olek, funktsiooni nimetus skriptikeeles ja viit C funktsioonile [9].

Ekspordi funktsioonide õige kasutamine on ennekõike programmeerija mure. Lua ei ole teadlik, kui palju parameetreid võib funktsioonile skriptist saata. Kõik parameetrid pannakse Lua kaitstud pinu osasse (Lua kasutab andmevahetuseks pinu, mis eraldab C funktsiooni kutsumiseks temale vajamineva osa) ning see on programmeerija ülesanne neid sealt õigesti lugeda ja eemaldada. Tagastatav(ad) operand(id) lükatakse tagasi pinusse ja C funktsioon peab kindlasti tagastama lükatud operandide arvu, et Lua API oskaks pinu korras hoida [9].

Skriptis defineeritud funktsioonide kutsumiseks ehk impordiks kasutatakse Lua pinu. Esimesena lükatakse pinusse funktsiooni nimi, millele järgnevad parameetrid nende defineerimise järjekorras skriptis. Peale funktsiooni kutseks valmistumist kutsutakse skriptis olev funktsioon käsuga *lua_call(lua_State*, int, int),* kus viimased 2 parameetrit on funktsiooni sisestatud parameetrite arv ja oodatavate tagastatavate väärtuste arv. Tagastatud tulemused tuleb minimaalsel juhul lihtsalt eemaldada pinust (Lua ei tee seda automaatselt). Enne on soovitatav väärtused lugeda, kasutades erinevaid Lua API poolt pinust lugemise funktsioone [9]. Joonis 5 näitab funktsioonide eksporti ja importi korrutamise näitel.

```c
// example.c
#include <stdio.h>
#include <lua.h>

// korrutab 2 täisarvu, tagastab tulemi
int mul(lua_State* pState)
{
    int a = (int)lua_tonumber(pState, 1);
    int b = (int)lua_tonumber(pState, 2);
    int c = a * b;
    lua_pushnumber(pState, c);
    return 1; // tagastab ainult ühe parameetri
}

int main(void)
{
    lua_State * pState = lua_open(1024); // 1024 = pinu suurus
    lua_register(pState, "multiply", mul); // ekspordi registreerimine
    // laadimine
    lua_dofile(pState, "example.lua");
    // globaalse muutuja X kirjutamine X = 256
    lua_pushnumber(pState, 10);
    lua_setglobal(pState, "X");

    // funktsiooni importimine
    lua_getglobal(pState, "callmul");
    lua_pushnumber(pState, 12);
    lua_call(pState, 1, 0);

    // globaalse muutuja X lugemine
    lua_getglobal(pState, "X");
    int X = (int)lua_tonumber(pState, 1);
    lua_pop(pState, 1); // pinu puhastamine
    printf("%i\n", X); // prindib tulemuse

    lua_close(pState); // lõpeta töö skriptiga
    return 0;
}
```

```lua
-- example.lua
X = 10
function callmul (Y)
    X = multiply(X, Y) -- kutsub C funktsiooni
end
```

Joonis 5. Funktsioonide export ja import kasutades Lua API

Lua API toetab lisaks veel skriptikeeles defineeritud globaalsete muutujate lugemist ja kirjutamist, et tagada programmi kontroll kõrgemal tasemel. Selle võib muidugi asendada *getter*/*setter* funktsioonidega skriptis [9].

2.3.2 Python

Python kasutab C/C++ ühenduse loomiseks Python/C API't, millel puudub aktiivne töötluskeskkond (*runtime environment*), nagu Lua puhul *lua state*. Kogu suhtlus skripti ja C vahel käib läbi Python'i objektide [9].

Python'i objekt – *PyObject* (edaspidi: objekt) – on igasugune infomatsioon, mis võib endas sisaldada muutujaid, funktsioone, mooduleid või isegi tervet skripti koodi. API

haldab objektide eluiga automaatselt ja programmeerija kasutab alati ainult nende viitasid. Iga Python'i objekti kasutuse kohta peetakse arvet ja kui programmeerijal ei ole enam vaja antud objekti kasutada, tuleb sellest API kaudu märku anda, kasutades funktsiooni *Py_XDECREF(PyObject*),* mis vähendab objekti kasutajate arvu. Objekt eemaldatakse automaatselt, kui sellele ei leidu enam ühtegi kasutajat [9], [16].

Skripti laadimiseks kasutatakse funktsiooni *PyImport_Import(PyObject*),* mis võtab ainsa parameetrina sisse objekti, mis sisaldab endas skriptifaili nime. Selle loomiseks kasutatakse funktsiooni *PyString_FromFile(const char*).* Mõlemad funktsioonid tagastavad objekti. Importimise puhul on tagastatava objekti sisuks moodul (mooduliks on kood koos kõigi muutujate, funktsioonide jt. Python'i koodis sisalduvate osadega). Esmasel laadimisel töödeldakse skriptis globaalselt kättesaadav kood koheselt [9].

C koodis defineeritud funktsioonide kutsumiseks Python'i skriptist tuleb pakkida C funktsioonid skriptis imporditavaks mooduliks. Kõik C funktsioonid, mida skriptis kutsutakse, peavad vastama prototüübile *PyObject* nimi(PyObject*, PyObject*)* ja tuleb registreerida mooduli funktsioonidena *PyMethodDef* massiivi kujul *{nimi, funktsiooni viit, lipud, dokumentatsioon}*. Lipud näitavad, millisel kujul parameetreid soovitakse (kasutatakse peaaegu alati METH_VARARGS). Oluline on, et funktsioonide deklaratsiooni massiiv lõppeks nö. null-funktsiooniga *{NULL, NULL, 0, NULL}*, et API tunneks massiivi lõpu ära. Uue mooduli loomine ja funktsioonide lisamine moodulile käib vastavalt kasutades funktsioone *PyImprt_AddModule(const char*)* ja *Py_InitModule(const char*, PyMethodDef*).* Näitena on toodud C funktsioon, mis teostab kahe arvu korrutamist (vaata Joonis 6) [9], [16].

Skriptis defineeritud funktsioonide kasutamiseks C koodis on vaja kasutada mooduli sõnastikku (*module dictionary*), mis sisaldab endas infot kõikide definitsioonide kohta (sealhulgas funktsioonide kohta). Esmalt on vaja leida objekt, mis sisaldab endas kutsutavat skripti funktsiooni. Selleks tuleb võtta sõnastik moodulist, kasutades *PyModule_GetDict(PyObject* moodul)* ja otsida sellest kutsutav funktsioon kasutades API funktsiooni *PyDict_GetItemString(PyObject* sõnastik, const char* nimi)*. Tulemuseks on objekt, mis sisaldab endas kõike vajalikku skripti funktsiooni kutsumiseks [9].

```c
// example.c
#include <stdio.h>
#include <Python.h>

/* pSelf - objekt, mis sisaldab endas skriptis kutsutavale funktsioonile nähtavat infot
   pParams - objekt, mis kannab endas parameetreid tuple andmetüübis */
PyObject* mul(PyObject* pSelf, PyObject* pParams)
{
    int a, b;
    // võta parameetrid. "ii" tähistab formaati. antud juhul loetakse kaks täisarvu: int, int
    PyArgParseTuple(pParams, "ii", &a, &b);
    return PyInt_FromLong(a * b); // tagastab objekti, milleks on täisarvuline korrutise tulemus
}

int main(void)
{
    Py_Initialize();
    // mooduli funktsioonid
    PyMethodDef CFunctions[] = {
        /*  skriptis kutsutava funktsiooni nimetus, C funktsioon,
            luba argumendid, dokumentatsioon (jäetud tühjaks) */
        {"multiply", mul, METH_VARARGS, ""},
        // NULL-funktsioon on vajalik, et API teaks millal definitsioonide lõpp on
        {NULL, NULL, 0, NULL}
    };
    // tee uus moodul
    PyImport_AddModule("CFun"); // mooduli nimi "CFun"
    Py_InitModule("CFun", CFunctions); // lisa C funktsioonid
    // skripti laadimine
    PyObject* pModule = PyImport_Import(PyString_FromString("example.py"));
    // skriptist funktsiooni kutsumine
    PyObject* pFun = PyDict_GetItemString(PyModule_GetDict(pModule), "callmul");
    PyObject* pParameters = PyTuple_New(1);
    PyTuple_SetItem(pParameters, 0, PyInt_FromLong(12)); // lisa parameeter 12
    int X = PyInt_AsLong(PyObject_CallObject(pFun, pParameters)); // kutsu funktsioon
    printf("%i\n", X);

    Py_Finalize();
    return 0;
}
```

```python
# example.py
import CFun

X = 10
# korruta kasutades C funktsiooni
def callmul(Y):
    return multiply(X,Y)
```

Joonis 6. Funktsioonide export ja import kasutades Python/C API

Kui skriptis defineeritud funktsioon vajab parameetreid, tuleb enne funktsiooni kutsumist panna kokku *tuple* objekt, kasutades *PyTuple_New(int)*, kus määratakse, mitu parameetrit skripti funktsioon vajab. Edasi tuleb juba kasutada *PyTuple_SetItem(PyObject* tuple, int parameetri_järjekord, PyObject* parameetri_väärtus)*, et täita *tuple* parameetritega, kusjuures parameetri väärtus peab olema Python'i objekt, mille loomiseks on mitmeid API andmeloomise funktsioone (näiteks täisarvu jaoks kasutada *PyObject* iParam = PyInt_FromLong(16)*, kus andmetüübi väärtuseks on 16) [9].

Viimaks saab kutsuda skriptis funktsiooni, kasutades *PyObject_CallObject(PyObject* funktsioon, PyObject* tuple)*, mis tagastab tulemusena objekti, mille sisuks on skriptifunktsiooni poolt tagastatav tulemus. Tulemuse lugemiseks C koodis tuleb see lahti pakkida vastavalt oodatavale andmetüübile (näiteks täisarvu saamiseks kasutada *int iRes = PyInt_AsLong(PyObject* tulemuse_objekt)* funktsioon). Täpsemalt on impordi protsess näidatud koos ekspordi näitega (vaata Joonis 6) [9].

3 Reaalajas arendamine SCR skriptikeeles

SCR on reaalajas arendustegevuseks sobiv programmeerimise tööriist. Antud tööriist on C/C++ programmis kasutatav liides ehk API, mis võimaldab liita skriptikeeles kirjutatud programmi süsteemiga arendustegevuse kiirendamise eesmärgil.

3.1 Ülevaade

Tööriista põhirõhk on pandud andmete väärtuste hoidmisele järjestikuste skriptiversioonide vahel. Skriptikeeles arenduse üheks oluliseks osaks on skripti laadimine ja testimine, kusjuures uuesti testitavasse olekusse viimine võib võtta palju aega. Seetõttu proovitakse uue skriptiversiooni laadimisel skriptimälus asuvate andmete väärtused jätta samaks, nagu eelmises versioonis.

SCR skriptikeelt ei sobi kasutada väärtuste hoidmise eesmärgil juhtudel, kui andmed kaotavad pidevalt oma kehtivuse. Andmete kehtivuse määrab toiming, kuidas andmete väärtus on saadud. Eelmises versioonis saadud andmete väärtused ei pruugi vastata enam reaalsetele tulemustele, kui hetkel arenduses olev skriptikeelne algoritm on otseselt vastutav mõnede andmete väärtuste arvutamise eest.

Skriptikeelel on assembler-laadne süntaks. Käsud tegelevad lihtsamate toimingutega, milleks on virtuaalmasina juhtimine ja andmevahetus mäluga. Selline kirjutusviis ei toeta reaalajas arendamise juures olulist koodikirjutamise kiirust. Seetõttu toetatakse koodi transleerimisel kõrgkeele filtrit, mis laseb defineerida skripti kirjutamise konkreetsele ülesandele sobiva kõrgkeelse süntaksi. Kuna SCR ei sea piiranguid konkreetsetele ülesannetele, kus seda kasutada, on kõrgkeele filtri kirjutamine tööriista kasutaja ülesanne.

3.2 API

SCR API pakub programmeerijale võimalust integreerida SCR skriptikeel oma C/C++ keeles kirjutatud programmiga võimalikult lihtsalt ja kiiresti. API jaguneb programmeerija poolt kasutatavateks andmestruktuurideks ja nendega seotud funktsioonideks. Lisaks andmestruktuuride ja funktsioonide tundmisele on tähtis tunda ka SCR virtuaalmasinat ja selle käsustikku.

3.2.1 Assembleri käsustik

SCR käsustikku kuuluvad juhtimis-, mälu-, aritmeetika- ja loogikakäsud (kokku 46 käsku). Virtuaalmasinal on pinu-mälu põhine arhitektuur. Seega on käsustiku käsud pinule orienteeritud ja tegelevad andmete liigutamisega pinu- ja andmemälu vahel.

Juhtimiskäsud on programmivoo juhtimiseks. Siia alla kuuluvad nii erinevad katkestuskäsud, koodiploki kutsed, kindla siirdumisega käsud ja tingimuslikud siirdekäsud. Mitmed käsud on teatud mõttes dubleeritud, sest dünaamilisuse eesmärgil on võimalik neid kutsuda nii ilmutatud kui ka ilmutamata operandidega.

Mälukäsud on andmete vahetamiseks virtuaalmasina pinumälu ja andmemälu vahel. Selleks, et teostada aritmeetika või loogika tehteid ja mõningaid juhtimiskäske, on vaja liigutada operandid pinusse. Sarnaselt on vaja töötluse tulemid viia tagasi andmemällu. Mälukäskude seas on nii ilmutatud kui ka ilmutamata aadressiga käske. Kõik aadressid on oma olemuselt täisarvulised andmetüübid ja seega võib aadressidega töötamiseks kasutada täisarvude jaoks mõeldud käske. Eraldi on küll käsud andmemuutuja aadressi ja koodiosa aadressi lükkamiseks pinusse, kuid edasi tulebki kasutada täisarvudele mõeldud käske.

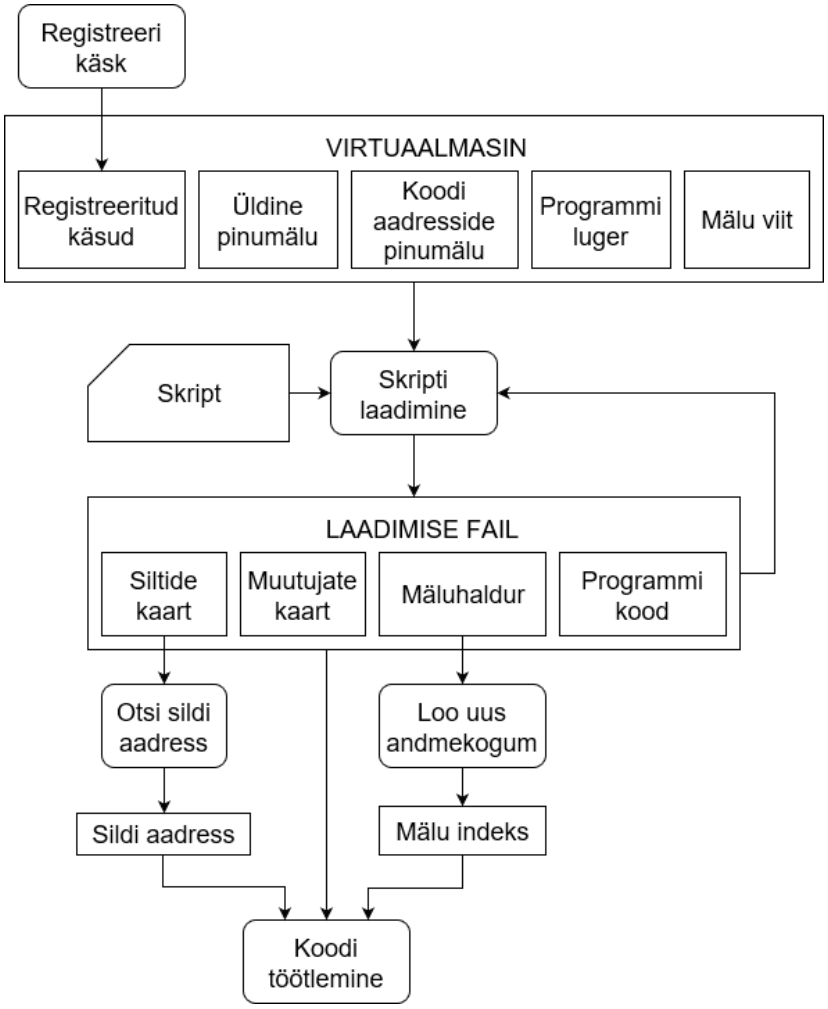
Kuigi tähed *b*, *i* ja *d* mälukäskude alguses on mõeldud, et juhtida tähelepanu, et käsud sobivad just vastavalt *boolean*, *integer* ja *double* andmetüüpide jaoks, on nende peamine erinevus siiski selles, et tegu on vastavalt 1-, 4- ja 8-baidiste kogumitega. Mälus ei ole keelatud kahe järjestikku asetseva 4-baidise täisarvu laadimist pinumälusse, kasutades käsku *dload*.

Aritmeetikakäsud on tehete teostamiseks pinus asuvate operandide peal. Operandid võetakse järjest pinu pealt ja tulem pannakse kõige viimaseks. Loogikakäsud on loogiliste tehete teostamiseks pinus asuvate operandidega. Tulemuseks on alati tõeväärtus. Loogikakäskude alla kuuluvad lisaks võrdlused andmetüüpide vahel. Antud käsud on vajalikud koos juhtimiskäskudega programmivoo juhtimisel.

3.2.2 Andmestruktuurid ja funktsioonid

SCR API kasutamiseks on oluline tunda ainult kahte andmestruktuuri: virtuaalmasinat (*ScrVM*) ja laadimise faili (*ScrLF*). Nende sees leidub rohkemgi andmestruktuure, millele programmist juurde pääseb, kuid nende kasutamiseks ja juurdepääsemiseks pakub API

juba mitmeid funktsioone. Kokku on 12 andmestruktuuri (vaata Joonis 11), kuid abstraktse mudelina saab esitada struktuuride vahelisi seoseid Joonis 7 abil.



Joonis 7. SCR API andmestruktuuride ja nende kasutamise abstraktne mudel

Virtuaalmasina ülesanneteks on hoida endas viiteid registreeritud käskudele ja töötluseks vajaminevaid komponente: programmi luger, viit töötlusel kasutatavale mälule, üldist andmetöötlusele mõeldud pinumälu ja koodi aadresside pinumälu. Iga registreeritud käsu kohta antakse käsule virtuaalmasina piires unikaalne käsukood ja SCR assembleri mnemoonika. Viimase määramine on programmeerija enda teha. Teisi virtuaalmasina komponente kasutatakse skriptikoodi töötluseks.

Skripti laadimise tulemus salvestatakse laadimise faili, mis sisaldab endas siltide ja muutujate kaarti. Kuigi laadimise fail käib ühe skripti kohta, võib ühe programmikoodi kohta olla mitu mälu. Nende haldamise eest vastutab mäluhaldur. Skripti laadimisel

kasutatakse eelnevat laadimise tulemust ja sellele mõeldud virtuaalmasinat. Samuti saab laadimise failist töötlemiseks vajamineva koodisildi aadressi ja indeksi mäluhalduri poolt eraldatud skripti mälule.

SCR API sisaldab 35 funktsiooni. Kasutuse sageduse ja olukorra järgi grupeeritakse need kahte klassi: ettevalmistamise ja töötluse funktsioonideks. Esimese klassi puhul kutsutakse funktsioone ainult programmi töö alguses ja lõpus. Teise klassi funktsioone kutsutakse peamiselt käskude poolt skriptikoodi interpreteerimise ajal.

Ettevalmistuse funktsioonide ülesandeks on ette valmistada andmestruktuure enne skriptikoodi töötlemist. Siinsed funktsioonid loovad ja puhastavad virtuaalmasina ning laadimise faili. Samuti kuulub siia alla funktsioon SCR assembleri käsu registreerimiseks virtuaalmasina jaoks ja skripti laadimise funktsioon (vaata Tabel 5).

Töötluse funktsioonide eesmärgiks on juhtida skriptikoodi töötlust. Siia alla kuuluvad kõik funktsioonid, mis aitavad programmeerijal kasutada virtuaalmasinat oma defineeritud käskudes (näiteks virtuaalmasina pinusse uue operandi lükkamine või sealt eemaldamine (vaata Tabel 6)).

3.2.3 Skriptikoodi laadimine

Skriptikoodi laadimine jaguneb viieks etapiks: skriptifaili sümbolite (*tokens*) laadimine; kõrgkeele filtreerimine; andmete ja siltide kaardistamine; andmete väärtuste säilitamine; transleerimine (Joonis 8).

Joonis 8. Laadimise etapid

Sümbolite (sõnad, *tokens*) eraldamise protsess muudab tekstina esitatud koodi lihtsamini töödeldavaks *string* massiiviks. Kõik sõnad, mis on eraldatud tühiku, reavahetuse või mõne muu nähtamatu sümboliga, moodustavad omaette *token*'i. Erandina eraldatakse veel omaette *token*'iks järgnevad sümbolid: **;.:{}[]()„“**. Kahe jutumärgi vahele jääv tekst moodustab üheainsa *token*'i. Lisaks arvestatakse ka *include* sõnaga märgitud lisafailidega. Kõik juurde lisatud failide sümbolid lisatakse juurde lisamise kohale. Kui

skripti tekstifailis esineb kommentaare (kujul: */* ... kommentaar ... */),* siis need eemaldatakse sümbolite eraldamise protsessis.

Kõrgkeele filtri eesmärgiks on võimaldada programmeerijal kirjutada skriptikoodi inimesele jaoks arusaadavama süntaksiga. Peale skriptifailist sõnade eraldamist saadetakse sõnad (*tokens*) selle olemasolul läbi filtri, mille ülesandeks on muuta sümbolite (*tokens*) massiiv translaatorile arusaadavaks kujuks. Kõrgkeele filtrit kasutades on oluline, et filtreerimise etappi tulemusena oleksid kõik sümbolitena esitatud käsud SCR assembleri kujul.

Andmete ja siltide kaardistamise käigus otsitakse sümbolite massiivist üles kõik mälupesade ja koodi siltide deklaratsioonid. Täpsemalt on sellest kirjutatud järgnevas punktis 3.2.4 Andmete haldamine. Koos siltide ja andmetega kaardistatakse ka transleerimisel vajaminevad käsud ja arvutatakse vajamineva koodi pikkus. Transleerimisel kirjutatakse koodi baidina esitatud käsu kood ja käsu juurde kuuluvate muutujate väärtused.

3.2.4 Andmete haldamine

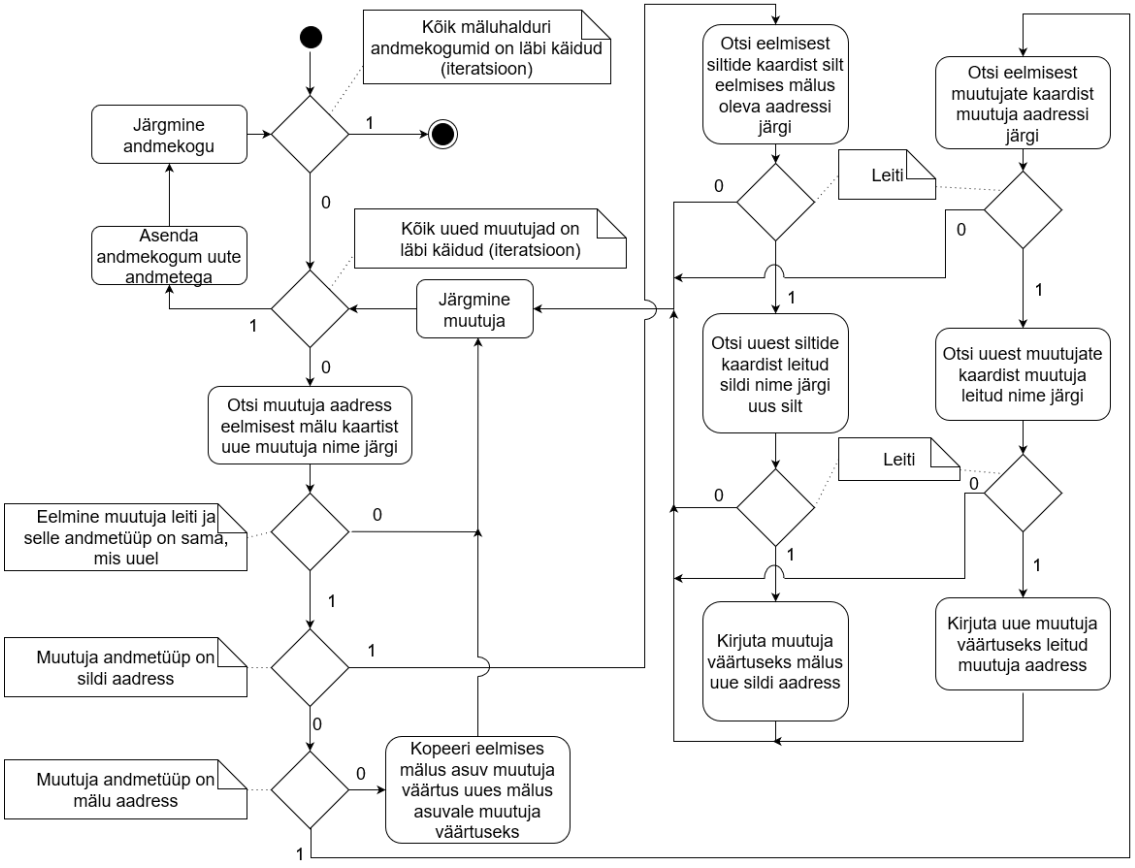
Ühe SCR skriptikeeles kirjutatud skriptikoodi kohta võib olla mitu samaselt adresseeritavat mälu. Mälude haldamise eest vastutab laadimise faili alla kuuluv mäluhaldur. Mäluhalduri ülesanneteks on eraldada laetud programmile vajaminev mälumaht ja hoida viidet sellele. Enne esmakordset skripti programmi töötlemist tuleb lasta mäluhalduril luua uus mälukogum, koodi uuesti samasse faili laadimisel selleks enam vajadust ei ole. Töötlemisel tuleb parameetrina määrata, mitmendat mälu soovitakse töötlusel kasutada.

Lisaks mälu haldamisele on mäluhalduri ülesandeks skriptifaili laadimisel eelnevalt laetud skripti mälus asuvate andmeväärtuste hoidmine. Kahes järjestikuses skriptiversioonis esinenud samanimeline muutuja saab uuemas versioonis oma väärtuseks viimase kooditöötluse lõppedes olnud väärtuse. Selle täitmiseks hoitakse laadimise failis andmete kaardil kõikide mälupesade aadresse koos skriptis deklareeritud nimede ja vastavate andmetüüpidega.

SCR toetab viit andmetüüpi: täisarv, koodisilt, mälu aadress, ujupunktarv ja tõeväärtus. Nendest esimesed kolm on 4-baidised andmetüübid ja käsitletakse kui tüüpilist C täisarvu

– *integer*. Ujupunktarv ja tõeväärtus on vastavalt 8-baidine *double* arv ja 1-baidine *char* (täheväärtus). Mälu andmete säilitamiseks on oluline deklareerida siltide ja aadresside hoidmiseks mõeldud mälupesad vastavalt nende mälupesade kasutamise eesmärgile, kuigi neid saab kasutada ka täisarvude hoidmiseks.

Iga skriptikoodi laadimisel kaardistatakse muutujad mälus, kusjuures muutujate järjekorra mälus määrab nende deklareerimise järjekord koodis. Kahe samanimelise muutuja tuvastamisel ignoreeritakse uusi samanimelisi muutujaid ehk kaardistatakse ainult esimene deklaratsioon. Eelneva skripti laadimise mälu kaardi tuvastamisel teostatakse peale uue kaardi loomist vana kaarti aluseks võttes muutujate väärtuste hoidmine (Joonis 9).



Joonis 9. Mälu oleku hoidmise algoritm

Kõik mäluhalduri poolt eraldatud andmekogumid asendatakse uue andmekogumiga, kus on võimalikult suures ulatuses säilitatud andmete väärtused. Laadimise käigus kaardistatud muutujate kohta otsitakse eelmisest kaardist muutuja nime järgi eelmine

muutuja. Selle leidmisel ja tüübi kokkulangemisel eeldatakse, et tegu on sama muutujaga ja eelneva muutuja väärtus kantakse üle uude mällu. Muutujate tüüpide (nagu *int*, *double* ja *bool*) juhul toimub üksühene väärtuste kopeerimine. Sildi (*label*) ja aadressi (*address*) juhul on tegemist viidaga vastavalt koodi aadressile ja mälu aadressile.

Sildi väärtuse alles hoidmiseks peab ka uues mälus olema muutuja väärtuse viit samale koodi aadressile, kusjuures aadress võib olla nüüd muutunud. Väärtuse asendamiseks õige sildi aadressiga on vaja ennekõike teada sildi nime, mis otsitakse eelnevast sildi kaardist aadressi järgi. Kui leitud nimele vastab ka uues koodis samanimeline silt, saab muutuja oma uueks väärtuseks leitud sildi aadressi.

Mälu aadressi juhul toimitakse sarnaselt siltide juhtumile. Erinevus seisneb selles, et otsimisi teostatakse mälu kaartidest.

3.3 SCR kasutamine

Skriptikeel SCR on kõige sobilikum suuremate ja mitmest abstraktsest komponendist koosnevate süsteemide arendamiseks. Sarnaselt kõigile teistele skriptikeeltele, mida kasutatakse reaalajas arendamiseks, pakub SCR skripti laadimist ning funktsiooni ekspordi ja impordi võimalusi.

SCR skripti laadimine käib kasutades funktsiooni *scrLoad*. Oluline erinevus teiste skriptide laadimisel on see, et kui kompileerimise fail sisaldab juba kompileerimise tulemust, siis proovitakse säilitada andmete olekud võimalikult suures ulatuses.

Funktsiooni ekspordiks on vaja SCR käsu prototüübile vastav funktsioon registreerida virtuaalmasina jaoks enne skripti kompileerimist. Selleks on ettenähtud funktsioon nimega scrRegisterInstruction(const char* mnemoonika, ScrFunction funktsioon, ScrVM* virtuaalmasin), kus vastavalt esimene parameeter on assemblerkäsu nimetus koos parameetrite formaadiga. SCR käsu prototüüp - ScrFunction – on kujul void nimi(ScrVM* virtuaalmasin).

Käsus ilmutatud parameetrid pannakse nende registreerimise järjekorras programmi koodi ja nende õigesti väljavõtmine funktsioonis on (rangelt) programmeerija ülesanne. Sellega tagatakse programmilugeri viitamine järgneva käsu peale (vastasel juhul interpreteeritakse järgneva parameetri esimest baiti käsukoodina ja terve programm läheb

nihkesse). SCR API pakub selleks funktsioone *scrParseInt*, *scrParseDouble* ja *scrParseChar*. Kõik need funktsioonid võtavad ainsa parameetrina sisse virtuaalmasina ja tagastavad vastavalt *int*, *double* või *char* tüüpi andmed. Ilmutamata andmete vahetuseks kasutatakse virtuaalmasina pinu koos *push/pop* käskudega.

SCR skriptikeeles on asendatud klassikaline funktsiooni import sildiaadressi impordiga. Põhjuseks on see, et SCR on ennekõike assembler-laadne keel, kus puuduvad funktsioonid nende klassikalises mõttes. Sildi aadress näitab ära, kust alustada koodi töötlust, kuid ei näita ära nende lõppu ehk puudub funktsiooni ala.

Importimine on hädavajalik, et üldse oleks võimalik skriptikoodi käivitada. Skripti laadimisel koodi töötlust ei toimu. Koodiaadressi saamiseks tuleb kasutada *scrFindLabel*, mis tagastab koodis täisarvuna kohaliku koodiaadressi (nihe programmikoodi algusest). Koodi töötluseks tuleb kasutada API funktsiooni *scrExecute*, kusjuures andmete indeks tuleb küsida esmasel koodi laadimisel, kasutades funktsiooni *scrGetDataInstance*, mis tagastab uute andmete järjekorranumbri andmehalduse süsteemis. Impordi ja ekspordi näitena on toodud punkti 2.3 all kirjeldatud korrutamise näidet (Joonis 10).

```c
#include "SCR.h"

void multiply(ScrVM* vm)
{
    int a = scrPopInt(vm);
    int b = scrPopInt(vm);
    scrPushInt(a*b, vm);
}

int main(void)
{
    ScrVM vm = scrCreateVirtualMachine(1024);
    scrRegisterInstruction("multiply;", multiply, &vm);

    ScrLF lf = scrCreateLoadFile();
    int dataindex = scrCreateDataInstance(&lf);
    scrLoad("example.scr", &lf, &vm, 0);

    int init = scrFindLabel("init", &lf);
    int callmul = scrFindLabel("callmul", &lf);
    scrExecute(init, dataindex, &lf); // initialiseeri

    // kutsu funktsioon ühe parameetriga
    scrPushInt(12, &vm);
    scrExecute(callmul, dataindex, &lf);

    scrClearLoadFile(&lf);
    scrClearVirtualMachine(&vm);

    return 0;
}
```

```
/* example.scr */
init:
    int X; ipush 10; istore X;
hlt;

callmul:
    iload X;
    multiply;
    istore X;
hlt;
```

Joonis 10. Funktsiooni eksport ja import kasutades SCR API't. Skriptikeeles defineeritud muutuja korrutatakse läbi C koodi poolt saadetud parameetriga kasutades eksporditud funktsiooni *multiply*.

SCR pakub võimalust defineerida ülesandele orienteeritud süntaks kasutades laadimisel filtrit, mis peab vastama funktsiooni prototüübile kujul *void filtri_nimi(ScrStringArray* tokens)*. Filtri eesmärgiks on muuta skriptifailist loetud kõrgkeele sõnad SCR assembleri kujule. Soovitatav on filtri sisendina tulev struktuur täielikult asendada uue samaväärse struktuuriga. Programmeerijat abistavad API's sisalduvad funktsioonid *scrCreateStringArray*, *scrClearStringArray* ja *scrPushString*. Filter ise kujutab oma olemuselt kompilaatorit ja lisaks kõrgkeele tõlkimisele assemblerkujule on soovitatav rakendada ka koodioptimeeringuid. SCR API ei abista muul moel filtrite kirjutamisel ja kogu vastutus langeb programmeerija oskustel ja teadmistele kompilaatorit kirjutada. Täpsemalt vaata Lisa 4 – SCR filtrite kirjutamine.

Kokkuvõte

Bakalaureusetöö põhieesmärgiks oli tutvustada skriptikeelte kastuamist reaalajas arendustegevuses ja kiirendada arendusprotsessi uue skriptikeelse lahendusega.

Töö tulemusena valmis reaalajas arendustööks sobiv tööriist – SCR. Lahendus võimaldab programmeerijal kirjutada skriptikeelt ülesande jaoks sobiva süntaksiga, kasutades ise defineeritud filtreid, samuti säilitatakse SCR skriptikeeles kirjutatud programmis esinenud muutujate väärtused kahe järjestiku töödeldud skriptiversiooni vahel.

SCR kiirendab reaalajas arendustegevust juhul, kui skriptiversioonide vahel on vaja hoida alles andmete väärtusi ja/või spetsiifilise ülesande täitmiseks on vaja defineerida mugavam süntaks. Lahenduse ühendamine ja ettevalmistus C/C++ programmis kasutamiseks nõuab ülesandele orienteeritud süntaksi filtri vajadusel siiski palju aega ja ilma selleta on võimaldatud ainult SCR assembleris skripti kirjutamine.

Kasutatud kirjandus

[1] M. Rouse, „What is scripting language,“ [Võrgumaterjal]. Available: http://searchwindevelopment.techtarget.com/definition/scripting-language. [Kasutatud 6 Märts 2017].

[2] A. Kanavin, „An overview of scripting languages,“ Lappeenranta, 2002.

[3] „techopedia,“ [Võrgumaterjal]. Available: https://www.techopedia.com/definition/19608/glue-language. [Kasutatud 6 Märts 2017].

[4] „The Programming Language Lua,“ PUC RIO, 7 Veebruar 2017. [Võrgumaterjal]. Available: https://www.lua.org/start.html. [Kasutatud 09 Aprill 2017].

[5] R. Lerusalimschy, Programming in Lua, Fourth edition, Lua.org, 2016.

[6] R. Lerusalimschy , Programming in Lua 3rd edition, Lua.org, 2013.

[7] tutorialspoint, „Lua - Basic Syntax,“ tutorialspoint, [Võrgumaterjal]. Available: https://www.tutorialspoint.com/lua/lua_basic_syntax.htm. [Kasutatud 22 Aprill 2017].

[8] Wikibooks, „Lua Programming/whitespace,“ Wikibooks, 14 Jaanuar 2015. [Võrgumaterjal]. Available: https://en.wikibooks.org/wiki/Lua_Programming/whitespace. [Kasutatud 22 Aprill 2017].

[9] A. Varanese, Game scripting mastery, Ohio: Stacy L. Hiquet, 2003.

[10] lua-users.org, „Lua Uses,“ lua-users.org, 6 Jaanuar 2017. [Võrgumaterjal]. Available: http://lua-users.org/wiki/LuaUses. [Kasutatud 22 Aprill 2017].

[11] Python Software Foundation, „What is Python? Executive Summary,“ Python Software Foundation, [Võrgumaterjal]. Available: https://www.python.org/doc/essays/blurb/. [Kasutatud 22 Aprill 2017].

[12] M. Lutz, Learning Python, 3rd Edition, O'Reilly Media, 2007.

[13] Python Software Foundation, „Sets - Unordered collections of unique elements,“ Python Software Foundation, [Võrgumaterjal]. Available: https://docs.python.org/2/library/sets.html. [Kasutatud 22 Aprill 2017].

[14] Python Software Foundation, „Applications for Python,“ Python Software Foundation, [Võrgumaterjal]. Available: https://www.python.org/about/apps/. [Kasutatud 22 Aprill 2017].

[15] S. Ferg, „Event-Driven Programming: Introduction, Tutorial, History,“ 2006.

[16] Python Software Foundation, „Extending Python with C or C++,“ 27 Märts 2017. [Võrgumaterjal]. Available: https://docs.python.org/2/extending/extending.html. [Kasutatud 30 Aprill 2017].

[17] „The Falcon Programming Language,“ [Võrgumaterjal]. Available: http://falconpl.org/. [Kasutatud 6 Märts 2017].

[18] „Techopedia,“ [Võrgumaterjal]. Available: https://www.techopedia.com/definition/26982/scripting-engine. [Kasutatud 6 Märts 2017].

[19] V. Beal, „webopedia,“ QuinStreet Enterprise, [Võrgumaterjal]. Available: http://www.webopedia.com/TERM/M/markup_language.html. [Kasutatud 19 Märts 2017].

[20] [Võrgumaterjal]. Available: http://ryanstutorials.net/bash-scripting-tutorial/bash-script.php. [Kasutatud 19 Märts 2017].

[21] P. Graham, On Lisp, Prentice Hall, 1993.

[22] Tutorialspoint, „Tutorialspoint,“ Tutorialspoint, [Võrgumaterjal]. Available: http://www.tutorialspoint.com/lisp/. [Kasutatud 10 Aprill 2017].

[23] realtimelogic, „Lua-FastTracks-Embedded-Web-Application-Development,“ realtimelogic.com, [Võrgumaterjal]. Available: https://realtimelogic.com/blog/2012/08/Lua-FastTracks-Embedded-Web-Application-Development. [Kasutatud 11 Aprill 2017].

[24] chaiscript.com, „ChaiScript - Easy to use scripting for C++,“ chaiscript.com, [Võrgumaterjal]. Available: http://chaiscript.com/. [Kasutatud 22 Aprill 2017].

[25] J. Turner ja J. Turner, „ChaiScript: readme,“ [Võrgumaterjal]. Available: https://codedocs.xyz/ChaiScript/ChaiScript/md_readme.html. [Kasutatud 22 Aprill 2017].

[26] J. Du, „Embedding Python in C/C++,“ CodeProject, 5 Oktoober 2005. [Võrgumaterjal]. Available: https://www.codeproject.com/Articles/11805/Embedding-Python-in-C-C-Part-I. [Kasutatud 24 Aprill 2017].

[27] D. M. Beazley , „SWIG : An Easy to Use Tool for Integrating Scripting Languages with C and C++,“ Department of Computer Science, 1996. [Võrgumaterjal]. Available: http://www.swig.org/papers/Tcl96/tcl96.html. [Kasutatud 24 Aprill 2017].

[28] A. Fedotov, „C++ embedded scripting: ChaiScript recipes,“ 7 Juuli 2015. [Võrgumaterjal]. Available: http://f5v.me/chaiscript-recipes. [Kasutatud 25 Aprill 2017].

Lisa 1 – SCR assembleri käsustik

Tabel 1. Juhtimiskäsud

käsukood	**mnemoonika**	**kirjeldus**
0x00	hlt;	peatab programmi töötluse.
0x01	jmp %l;	viib programmi töötluse uuele koodi aadressile, mis on märgistatud sildiga
0x02	jmpa;	viib programmi töötluse uuele koodi aadressile, mis on pinu peal
0x03	jmpif %l;	viib programmi töötluse uuele koodi aadressile, mis on märgistatud sildiga ainult siis, kui pinu peal olev väärtus on tõene (ehk mitte-nulline)
0x04	jmpifa;	viib programmi töötluse uuele koodi aadressile, mis on pinu peal ainult siis, kui aadressile eelnev väärtus pinus on tõene (ehk mitte-nulline)
0x05	call %l;	viib programmi töötluse uuele koodi aadressile, mis on märgistatud sildiga ning lükkab koodiaadressi pinusse peale *call* käsku täitmisele tuleva koodiaadressi
0x06	calla;	viib programmi töötluse uuele koodi aadressile, mis on pinu peal ning lükkab koodiaadressi pinusse peale *call* käsku täitmisele tuleva koodiaadressi
0x07	ret;	viib programmi töötluse aadressile, mis asub koodiaadressi pinu peal; eemaldab koodiaadressi pinust selle aadressi

Tabel 2. Mälukäsud

käsukood	**mnemoonika**	**kirjeldus**
0x08	iload %a;	laeb andmemälust 4-baidise operandi aadressilt, mis on käsus ilmutatult
0x09	iloada;	laeb andmemälust 4-baidise operandi aadressilt, mis on pinu peal
0x0A	istore %a;	salvestab pinu tipus asuva 4-baidise operandi andmemällu aadressile, mis on käsus ilmutatult
0x0B	istorea;	salvestab andmemällu pinu tipus asuvale aadressile talle pinus järgneva 4-baidise operandi

0x0C	ipush %i;	pinu tippu pannakse 4-baidine täisarv, mis on esitatud käsukoodi järel
0x0D	ipop;	pinu tipust eemaldatakse 4 baiti
0x0E	dload %a;	laeb andmemälust 8-baidise operandi aadressilt, mis on käsus ilmutatult
0x0F	dloada;	laeb andmemälust 8-baidise operandi aadressilt, mis on pinu peal
0x10	dstore %a;	salvestab pinu tipus asuva 8-baidise operandi andmemällu aadressile, mis on käsus ilmutatult
0x11	dstorea;	salvestab andmemällu pinu tipus asuvale aadressile talle pinus järgneva 8-baidise operandi
0x12	dpush %d;	pinu tippu pannakse 8-baidine ujupunktarv, mis on esitatud käsukoodi järel
0x13	dpop;	pinu tipust eemaldatakse 8 baiti
0x14	bload %a;	laeb andmemälust 4-baidise operandi aadressilt, mis on käsus ilmutatult
0x15	bloada;	laeb andmemälust 1-baidise operandi aadressilt, mis on pinu peal
0x16	bstore %a;	salvestab pinu tipus asuva 1-baidise operandi andmemällu aadressile, mis on käsus ilmutatult
0x17	bstorea;	salvestab andmemällu pinu tipus asuvale aadressile talle pinus järgneva 1-baidise operandi
0x18	bpush %b;	pinu tippu pannakse 1-baidine tõeväärtus, mis on esitatud käsukoodi järel
0x19	bpop;	pinu tipust eemaldatakse 1 bait
0x1A	i2b;	pinu tipus asuv 4-bainine operand teisendatakse tõeväärtuseks; tulemuseks on 1-baidine operand, mis on mitte-nulline, kui teisendatav ei olnud mitte-nulline
0x1B	i2d;	pinu tipus asuv 4-baidine operand teisendatakse 8-baidiseks ujupunktarvuks; pinu pealmised 4 baiti asendatakse 8 baidiga
0x1C	d2i;	pinu tipus asuv 8-baidine operand teisendatakse 4-baidiseks täisarvuks; pinu pealmised 8 baiti asendatakse 4 baidiga
0x1D	apush %a;	pinu tippu pannakse 4-baidine täisarvust mäluaadress, mis on esitatud käsukoodi järel; käsus kasutatakse mäluaadressi nime
0x1E	lpush %l;	pinu tippu pannakse 4-baidine täisarvust koodiaadress, mis on esitatud käsukoodi järel; käsus kasutatakse koodi silti

Tabel 3. Aritmeetika käsud

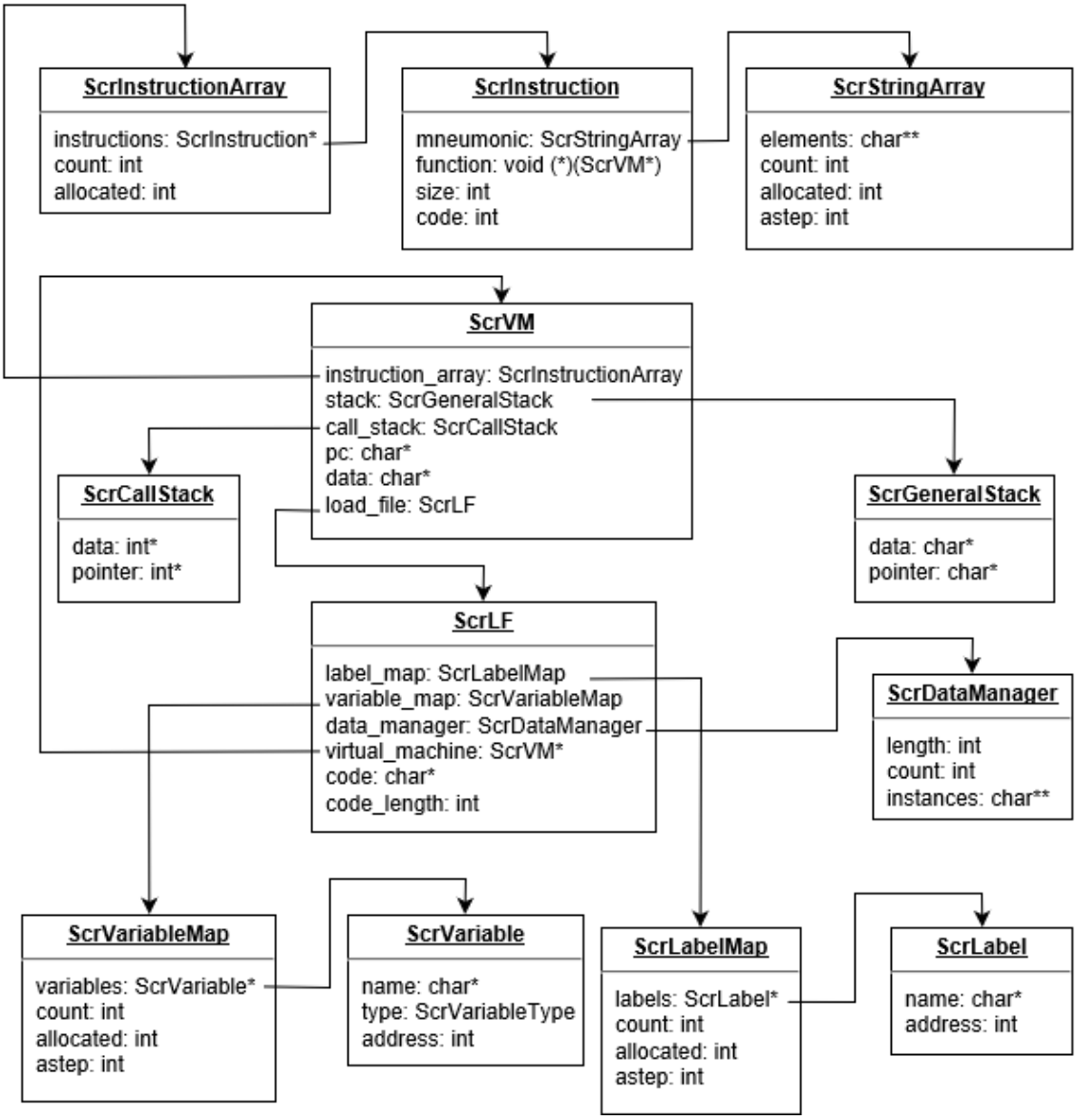
käsukood	**mnemoonika**	**kirjeldus**
0x1F	iadd;	kaks järjestikku pinumälu peal asuvat 4-baidist operandi liidetakse kui täisarve; tulem pannakse pinu peale
0x20	isub;	kaks järjestikku pinumälu peal asuvat 4-baidist operandi lahutatakse kui täisarve, vastavalt a – b, kus b on pinus kõige pealmine operand; tulem pannakse pinu peale
0x21	imul;	kaks järjestikku pinumälu peal asuvat 4-baidist operandi korrutatakse kui täisarve; tulem pannakse pinu peale
0x22	idiv;	kaks järjestikku pinumälu peal asuvat 4-baidist operandi jagatakse kui täisarve, vastavalt a / b, kus b on pinus kõige pealmine operand; tulem pannakse pinu peale
0x23	imod;	kaks järjestikku pinumälu peal asuvat 4-baidist operandi jagatakse kui täisarve, vastavalt a / b, kus b on pinus kõige pealmine operand; tulemiks on jagamise täisarvuline jääk, pannakse pinu peale
0x24	dadd;	kaks järjestikku pinumälu peal asuvat 8-baidist operandi liidetakse kui ujupunktarve, tulem pannakse pinu peale
0x25	dsub;	kaks järjestikku pinumälu peal asuvat 8-baidist operandi lahutatakse kui ujupunktarve, vastavalt a – b, kus b on pinus kõige pealmine operand, tulem pannakse pinu peale
0x26	dmul;	kaks järjestikku pinumälu peal asuvat 8-baidist operandi korrutatakse kui ujupunktarve; tulem pannakse pinu peale
0x27	ddiv;	kaks järjestikku pinumälu peal asuvat 8-baidist operandi jagatakse kui ujupunktarve, vastavalt a / b, kus b on pinus kõige pealmine operand; tulem pannakse pinu peale

Tabel 4. Loogikakäsud

käsukood	**mnemoonika**	**kirjeldus**
0x28	inv;	pinu peal asuv 1-baidine operand inverteeritakse kui tõeväärtus, mitte-nullisest väärtusest saab nulline ja vastupidi; tulem pannakse pinu peale
0x29	and;	kahe järjestiku 1-baidise operandi vahel teostatakse loogiline JA tehe; tulem pannakse pinu peale
0x2A	or;	kahe järjestiku 1-baidise operandi vahel teostatakse loogiline VÕI tehe; tulem pannakse pinu peale
0x2B	icmp;	kahe järjestiku 4-baidise operandi vahel teostatakse võrdlustehe, tulemiks on 1-baidine tõeväärtus, mis on mitte-

		nulline, kui operandid olid võrdsed. Tulem pannakse pinu peale
0x2C	icmpgte;	kahe järjestiku 4-baidise operandi vahel teostatakse võrdlustehe, tulemiks on 1-baidine tõeväärtus, mis on mitte-nulline, kui pinu peal asuv operand oli suurem või võrdne; tulem pannakse pinu peale
0x2D	dcmp;	kahe järjestiku 8-baidise operandi vahel teostatakse võrdlustehe, tulemiks on 1-baidine tõeväärtus, mis on mitte-nulline, kui operandid olid võrdsed; tulem pannakse pinu peale
0x2E	dcmpgt;	kahe järjestiku 8-baidise operandi vahel teostatakse võrdlustehe, tulemiks on 1-baidine tõeväärtus, mis on mitte-nulline, kui pinu peal asuv operand oli suurem või võrdne; tulem pannakse pinu peale

Lisa 2 – Andmestruktuurid



Joonis 11. Andmestruktuurid ja nendevahelise ühendused

Lisa 3 – Funktsioonid

Tabelid ettevalmistuse (Tabel 5) ja töötluse (Tabel 6) funktsioonidele. Kõik funktsioonid, mis tagastavad midagi, sisaldavad kirjelduses tagastatava andmetüübi nimetust.

Tabel 5. SCR API ettevalmistuse funktsioonid

funktsiooni nimi	**parameetrid**	**kirjeldus**
scrCreateVirtualMachine	const int – üldise pinumälu suurus baitides. Väärtuse 0 korral kasutatakse vaikimisi suurust 1024 baiti.	tagastab kasutamiseks valmis virtuaalmasina - ScrVM
scrClearVirtualMachine	ScrVM* – viit virtuaalmasinale	puhastab virtuaalmasina ja viib selle algolekusse
scrRegisterInstruction	const char* – käsu mnemoonika	registreerib funktsiooni, mida virtuaalmasin saab kasutada
	ScrInstructionFunction – viit käsu funktsioonile	
	ScrVM* - viit virtuaalmasinale, mille jaoks käsk registreeritakse	
scrCreateLoadFile	-	tagastab kasutamiseks valmis laadimise faili - ScrLF
scrClearLoadFile	ScrLF* - viit laadimise failile	puhastab laadimise faili ja viib selle algolekusse
scrLoad	const char*- skriptifaili nimi	laeb skriptifaili laadimise faili; kaardistab sildid ja muutujad; transleerib SCR assembleri virtuaalmasina jaoks baidikoodi
	ScrLF* - viit laadimise failile, kuhu salvestatakse tulem	
	ScrVM* - viit virtuaalmasinale, millel peab kood töötama	
	ScrFilter – eelfilter	
scrCreateStringArray	int – sammu suurus, kui palju korraga mälu juurde võetakse (kui jääb puudu)	loob uue dünaamilise *string* massiivi, mis saab endas hoida *string* andmetüüpe – ScrStringArray (laadimise filtris kasutamiseks)
scrClearStringArray	ScrStringArray* viit *string* massiivile	puhastab dünaamilise *string* massiivi

		(laadimise filtris kasutamiseks)
scrPushString	const char* - *string*	lükkab dünaamilisse *string* massiivi uue *string*i; haldab automaatselt mälu, kui sellest jääb puudu
	ScrStringArray* - viit dünaamilisele *string* massiivile	

Tabel 6. SCR API töötluse funktsioonid

funktsiooni nimi	**parameetrid**	**kirjeldus**
scrFindLabel	const char* - sildi nimi	otsib kompileeritud koodist sildile vastava koodiaadressi ja tagastab selle täisarvuna – int; kui koodi silti ei leitud tagastatakse väärtus -1
	ScrLF* - laadimise faili (tulemuse) viit	
scrFindLabelByAddress	const int - aadress	otsib sildi siltide kaartist aadressi järgi
	ScrLabelMap* - siltide kaart	
scrCreateDataInstance	ScrLF* - laadimise faili (tulemuse) viit	loob uue andmekogumi ja tagastab selle indeksi täisarvulise väärtusena - int
scrExecute	int – koodi aadress, millest alates koodi töödeldakse	töötleb koodi kompileerimise failis defineeritud virtuaalmasinal alates etteantud koodi aadressist
	int – andmete kogumi indeks, mida andmetena kasutatakse	
	ScrLF* - laadimise faili (tulemuse) viit	
scrHalt	ScrVM* - viit virtuaalmasinale	peatab koodi töötluse
scrSetPC	int – koodi aadress	muudab programmilugeri aadressi
	ScrVM* - viit virtuaalmasinale	
scrStoreInt	int – mälu aadress	salvestab 4-baidise täisarvulise väärtuse mällu
	int - väärtus	
	ScrVM* - viit virtuaalmasinale	
scrStoreDouble	int – mälu aadress	salvestab 8-baidise ujupunktarvu mällu
	double - väärtus	
	ScrVM* - viit virtuaalmasinale	
scrStoreChar	int – mälu aadress	salvestab 1-baidise väärtuse mällu
	char - väärtus	

	ScrVM* - viit virtuaalmasinale	
scrLoadInt	int – mälu aadress ScrVM* - viit virtuaalmasinale	tagastab 4 baidise täisarvu mälust - int
scrLoadDouble	int – mälu aadress ScrVM* - viit virtuaalmasinale	tagastab 8-baidise ujupunktarvu mälust - double
scrLoadChar	int – mälu aadress ScrVM* - viit virtuaalmasinale	tagastab 1-baidise väärtuse mälust - char
scrParseIntVM	ScrVM* - viit virtuaalmasinale	võtab koodist 4 järgnevat baiti ja tagastab need täisarvuna – int; suurendab programmilugerit 4 võrra
scrParseDoubleVM	ScrVM* - viit virtuaalmasinale	võtab koodist 8 järgnevat baiti ja tagastab need pika ujupunktarvuna – double; suurendab programmilugerit 8 võrra
scrParseCharVM	ScrVM* - viit virtuaalmasinale	võtab koodist järgneva baidi ja tagastab selle tähemärgina – char. Suurendab programmilugerit ühe võrra.
scrPushCall	ScrVM* - viit virtuaalmasinale	lükkab programmilugeris oleva väärtuse koodiaadressite pinu peale
scrPopCall	ScrVM* - viit virtuaalmasinale	eemaldab koodi aadresside pinu pealt aadressi väärtuse ja tagastab selle täisarvuna - int
scrPushInt	int – täisarvuline väärtus ScrVM* - viit virtuaalmasinale	lükkab andmepinusse 4-baidise väärtuse, mis on esitatud täisarvuna
scrPopInt	ScrVM* - viit virtuaalmasinale	eemaldab andmepinust 4 baidise väärtuse ja tagastab selle täisarvuna - int
scrPeekInt	ScrVM* - viit virtuaalmasinale	tagastab andmepinu peal oleva 4-baidise väärtuse täisarvuna - int
scrPushDouble	double – pika ujupunktarvu väärtus ScrVM* - viit virtuaalmasinale	lükkab andmepinusse 8-baidise väärtuse, mis on esitatud pika ujupunktarvuna

scrPopDouble	ScrVM* - viit virtuaalmasinale	eemaldab andmepinust 8-baidise väärtuse ja tagastab selle pika ujupunktarvuna - double
scrPeekDouble	ScrVM* - viit virtuaalmasinale	tagastab andmepinu peal oleva 8-baidise väärtuse pika ujupunktarvuna - double
scrPushChar	char – väärtus tähemärgina ScrVM* - viit virtuaalmasinale	lükkab andmepinusse 1-baidise väärtuse, mis on esitatud tähemärgina
scrPopChar	ScrVM* - viit virtuaalmasinale	eemaldab andmepinust 1-baidise väärtuse ja tagastab selle tähemärgina - char
scrPeekChar	ScrVM* - viit virtuaalmasinale	tagastab andmepinu peal oleva 1-baidise väärtuse tähemärgina - char

Lisa 4 – SCR filtrite kirjutamine

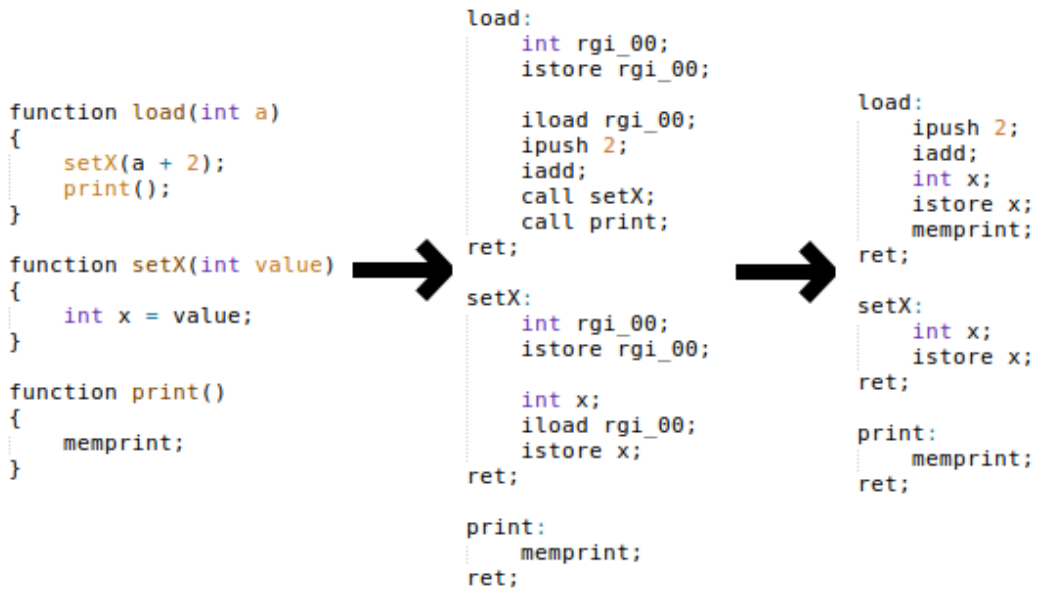
Filtrite kirjutamisel võimaldab programmeerijal kirjutada skriptikoodi kõrgemas keeles kui seda on SCR assembler. Tuleb tähele panna, et filtrite kirjutamine on keeruline ja aeganõudev tegevus. Isegi kõige lihtsamad kõrgkeele filtrid on 500 ja enam C koodirida.

```
void filter(ScrStringArray* tokens)
{
    // container to hold SCR assembly tokens
    ScrStringArray assembly = scrCreateStringArray(10);

    // parse through all tokens
    for (int i = 0; i < tokens->count; ++i) {
        ...
        // parse sentences
        ...
    }

    // replace tokens with assembly
    scrClearStringArray(tokens);
    tokens[0] = assembly;
}
```

Joonis 12. SCR filtri ülesehituse näide.



Joonis 13. Kõrgkeeles kirjutatud skriptikoodi (vasakul) põhjal genereeritakse temale vastav SCR assembler-koodiks (keskel). Ilma optimeeringuid rakendamata on genereeritud assembler-kood tunduvat suurem, kui käsitsi optimeeritud tulemus (paremal).