LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

PIRMKODA STATISTISKĀS ANALĪZES AUTOMATIZĀCIJA

BAKALAURA DARBS

Autors: Jānis Knets
Studenta apliecības Nr.: jk07108
Darba vadītājs: profesors Dr. Leo Seļāvo

RĪGA 2013

Anotācija

Šī dokumenta nolūks ir paskaidrot lietotājam par izveidoto programmatūtu, kā tā darbojas un viņas iekšējiem procesiem. Izmantotās tehnoloģijas ir LEX, YACC, C\C++, AVL koki., daudzpavedienu izstrāde. Komunikācija notiek ar ligzdu palīdzību. Programmatūra tikai strādā UNIX 64 bitu vidēs. Programmatūra ir veidota no trim lielākiem moduļu apkopojumiem, kuri, savukārt, veido tris binārās datnes.

Abstract

This document purpose is to describe created programms and processes within them to reader or potential user. Used technologies include, but are not limited to, LEX, YACC, C\C++. AVL trees, multithreading. For communication purpose UNIX sockets are used. Whole project so far should work only under 64bit unix machines. Programms consist out of modules that are gathered in 3 programms.

Atslēgvārdi

Moduļu bāzēta sistēma, pirmkoda pārveidošana, pirmkoda statistiskā analīze, algoritmu atpazīšana.

Saturs

Apzīmējumu Saraksts....6
1.Ievads....7
1.1.Mērķi....7
1.2.Aktualitāte....7
1.3.Darba struktūra....8
2.Bilde no augšas / Pamatprocesu apraksts....9
2.1.Eksistējošie moduļi....9
2.2.Moduļu sadalijums pa izpildāmām programmām....10
2.3.Skats uz datu plūsmu....11
3.Moduļi....13
3.1.Datu iegūšana no pirmkoda....13
3.2.LEX....13
3.2.1.LEX atslēgvārdi....13
3.3.LEX/YACC mijiedarbība....13
3.4.Komunikātors / “Communicator”....15
3.4.1.Ziņojumu veids....15
3.4.2.Ziņojumu struktūra....15
3.4.3.Tipiskas kumunikācijas piemēri....16
3.5.Pārveidotājs / “Transformer”....17
3.6. Datubāze / “Primal”....18
3.6.1.Nukleotīdu struktūra....18
3.6.2.Datu glabāšanas veids....20
3.6.3.Datu meklēšana....20
3.7.Analizētājs / “Analyser”....21
3.7.1.Pārkārtošanas prioritātes....21
3.7.2.Salīdzināšana....21
3.7.3.Analīzes rezultāti un potenciāls....22
3.8.Organisms / “Organism”....22
4.Datu apstrādeS CeļŠ....23
5.Salīdzinājums ar līdzīgu porgrammatūru....26
5.1.APTS....26
5.2.Cloc un līdzīgi....27
6.Rezultāti....28
7.Izmantotā literatūra....29
1.pielikums. LEX definīciju tabulas....30
2.pielikums. Nukleotīda struktūra....34
3.pielikums. Nukleotīda apakštipi....35
Dokumentārā lapa....38

APZĪMĒJUMU SARAKSTS

Nukleotīds – dotā dokumenta ietvaros tiek uzskatīts par datu apkopojumu, kuri kopā veido vienu ierakstu datubāzē. Reprezentē vienu darbību pirmkodā.

LEX – leksikas analizātors, ar šīs programmas palīdzību no pirmkoda tiek veidots atslēgvārdu kopums

YACC – Yet Another C Compiler, jeb Vēl viens C kompilātors. Programmatūra, kura izveido likumus pirmkoda pārstrādei.

AVL – Binārs pašbalansējošs koks. AVL ir izgudrotāju uzvārdu pirmie burti

BNF – Bakusa – Naura Forma, likumi ar kuriem bieži apraksta programmēšanas valodas sintaksi.

1. IEVADS

Dotais darbs apskatīs autora izveidotu programmu kopumu, kuru mērķis ir atpazīt C valodas pirmkoda piemērus un salīdzināt tos.

Pirmajā dokumenta daļā tiks izskaidrota kopējā arhitektūra un katrs modulis atsevišķi. Moduļu iekšējie procesi un dažās saistības ar citiem moduļiem.

Otrā dokumenta daļa stāsta par datu plūsmām caur programmu kopumu. Tiek izskaidrotas visas datu pārveidošanas. Īsumā – tiek iziets pilns datu cikls no padotā pirmkoda līdz izvadītiem rezultātiem.

Nobeigumā dotais risinājums tiks salīdzināts ar līdzīgiem, brīvi pieejamiem un tiks apspriesti dotā darba rezultāti un nākotne.

1.1. Mērķi

Šī dokumenta mērķis ir rast sapratni par iekšējo processu norisi, sniegt gan pamācoša rakstura informāciju, gan pamatu nākotnes attīstībai. Izstrādātās programmatūras[9] mērķis ir spēt saprast un novērtēt divus (vai vairākus) padotā pirmkoda piemērus. Salīdzināšana notiek gan rezultātu ziņā, gan paša pirmkoda līmenī – vai tas tika izmantots tas pats algoritms, bet ar pamainītu mainīgo deklarāciju.

1.2. Aktualitāte

Pirmkoda statiska analīze būs vajadzīga vienmēr. Tomēr tas prasa daudz laika un to ir ļoti grūti automatizēt. Sintaktiskās kļūdas, protams, datorprogrammas spēj atrast, bet valodas jēgu tās tikai mācās saprast.

Tācu šis dokuments nav par statisko analīzi, bet gan par statistiskās analīzes automatizāciju. Šādas programmatūras kopums dod iespēju pietiekoši ātri iegūt sākotnējo analīzi par izmantotiem algoritmiem. Labs pielietojums arī ir plaģiātu meklējumi.

Pirmkoda analīze būs nepieciešama vienmēr. Statistiskā anlīze ir jebkuras nozares pašsaprašanas rīks. Sākot ar jebkādu sistēmu būvi, mēs gribēsim to izprast un atrast tās potenciālās kļūdas. Ar katru gadu uzsvars uz datorprogrammatūru nepieiciešamību un to faktiskās

izmantošanas ikdienas processos palielinās. Šodien mēs nevarētu iedomāties kā būtu pirkt preces maksājot ar skaidru naudu un uz vietas nebūtu datorizēts kases aparāts. Sabiedriskajā transporta tiek izmantotas eletroniskas kartes kā biļetes. Bet analīze par izstrādātām programmām nav spērusi lielus soļus uz priekšu jau kādu laiku.[2]

1.3. Darba struktūra

Sākumā radās ideja par ģenētiskas un modulāras programmatūras izveidošanu, kura varētu būt spējīga pati attīstīties. Tas tika aprakstīts autora iepriekšējā rakstā "Pamatmodelis moduļu-bāzētai mākslīgai dzīvībai"[1] Balstoties uz paustām idejām minētā darbā tika sākts veidot sistēmu, kura būtu spējīga pati modificēt padoto pirmkodu. Pielāgot pirmkodu vajadzīgam risinājumam vai veidoti pati no pamatā pieejamās datubāzes.

Uz doto bridi tika izvēlēts mērķis iemācīt programmu atpazīt un saprast primkodu. Tas tiks panākts ar LEX un YACC[3], kuri dos iespēju analizēt jau eksistējošu pirmkodu. Tika izstrādāti likumi ar kuru palīdzību pirmkods tiek pārveidots par simboliskām virknēm, un vēlāk par datubāzes ierakstiem.

Jāpiezīmē ka dotā programmatūra nebūt nav tās pabeigtā stāvoklī un turpinās attīstību, līdz ar to var rasties dažas nesaskaņas salīdzinot šajā dokumentā rakstīto ar reālu pirmkodu.

2. BILDE NO AUGŠAS / PAMATPROCESU APRASKTS

Dotā nodaļa stāsta par katru moduli atsevišķi. Ikkatram modulim ir savs nolūks. Bet programmatūra ir spējīga darboties tikai šo moduļu ciešas sadarbības rezultātā. Tomēr šī modularitāte atstāj iespēju pievienot jaunus moduļus, kuri veiks papildus darbības. Tas ir iespējams, jo pamatā tiek izmantota vienota komunikācija ar iepriekš aprakstītu protokolu caur ligzdām(unix socket) (Protokols tiks aprakstītis pie komunikāciju moduļa kopējā apraskta.), visa komunikācija ir veicama caur iepriekš izveidotu moduli, kurā būtu viegli pievienot papildus funkcionalitāti, kā rezultātā tiek iegūta pietiekoši veikla un viegli lokāma sistēma dažādām vajadzībām. Tomēr ir vajadzīgas zināšanas, lai šos rīkus varētu effektīvi izmantot. Daļu no šīm zināšānām arī cenšas sniegt dotais dokuments.

2.1. Eksistējošie moduļi.

Moduļi, kuri tiek izmantoti izstrādātās programmatūrā uz dokumenta uzrakstīšanas brīdi ir šie:

1. LEX\YACC tulkotājs,
2. Pārveidotājs (Transformer),
3. Datu bāze (Primal),
4. Analizātors (Analayzer),
5. Tiesnesis (Judge),
6. Organisms (Organism),

kā arī eksīstē, bet netiek gluži izmantoti:

1. Pārraugs (Caregiver),
2. Sargsuns (Watchdog),
3. Mutētājs (Mutator).

Moduļi ir sakārtoti pēc datu virzības secības caur tiem.

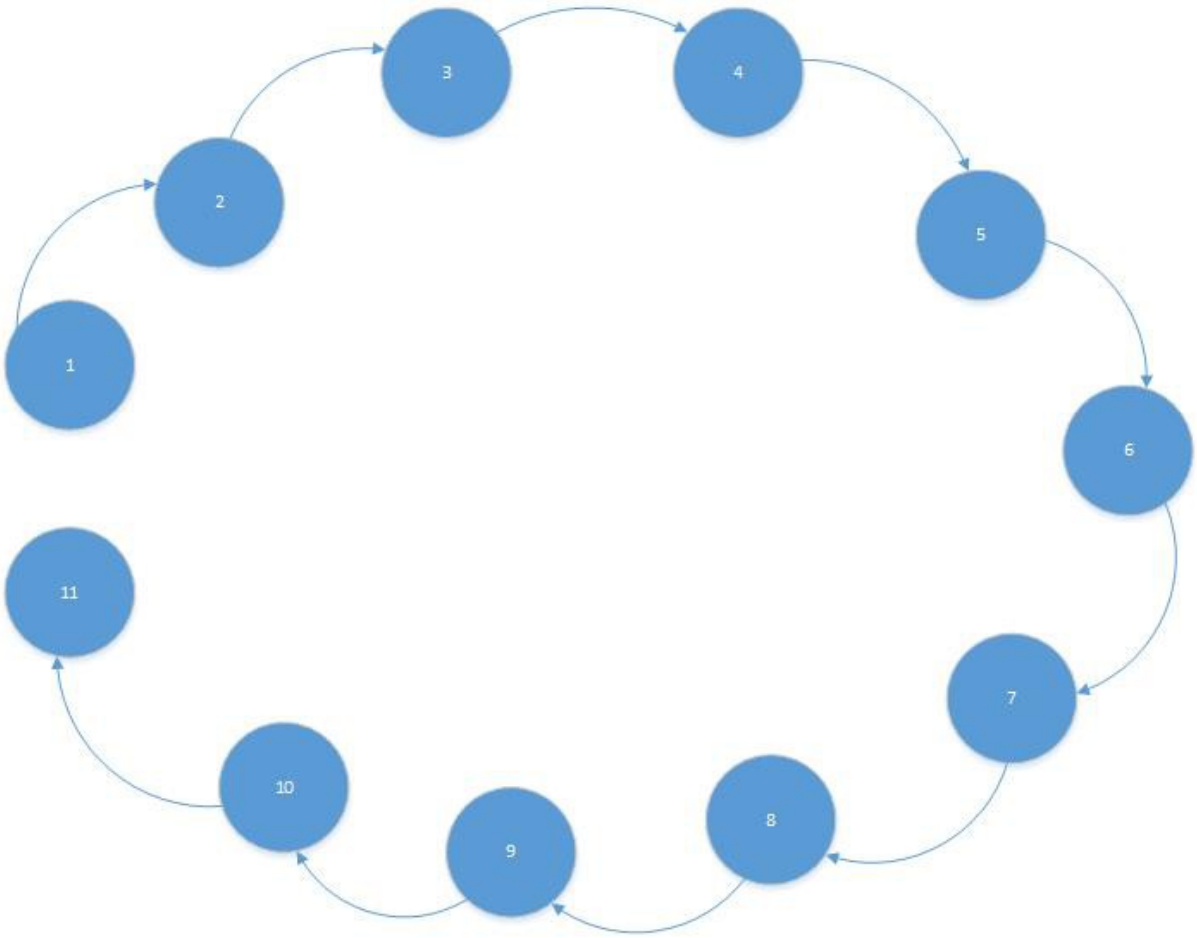
2.2. Moduļu sadalijums pa izpildāmām programmām

Pirmie seši minētie moduļi veido trīs programmatūras datnes -

1. lexer.bin – sastāv no LEX YACC ģenerētām datnēm. Bāzes likumu kopumus, kurš ļauj atpazīt pirmkodu, un pārveidot to Pasaulei saprotamos datos.
2. world.bin – jeb Pasaule, sastāv no visiem pārējiem moduļiem izņemot Organisms. Galvenais process, kurš pārrauga visu notiekošo, lielākā algoritmu daļa ir iekļauta tieši šajā daļā.
3. organism.bin, Organisms ir čaulas modulis ap padoto pirmkodu. Tas veic pirmkoda kompilēšanu, startēšanu un pēc pabeigšanas paziņo Pasaulei par darba beigām. Turpmākā izstrādē Organisma pārvaldē esošie moduļi varētu konkurēs par resursiem, tādā veidā atveidot cīņu par izdzīvošanu.

2.3. Skats uz datu plūsmu

Lai novērstu lasītāja nesaprašanu kādā veidā moduļu ir savstarpēji saistīti tiek dots īss apraksts par datu virzību caur visiem moduļiem datu plūsmas diagrammā.



att 2.1: Datu plūsma

1. Lietotājs startē world.bin un padod tai sākotnējo konfigurāciju.
2. Notiek inicializācija, startēts lexer.bin, kurš izveido savienojumu ar world.bin un gaida informāciju par apskatāmām pirmkoda datnēm.
3. Lietotne ievāc informāciju par padoto pirmkoda datņu nosaukumiem. Informācija var būt ievākt no iepriekš paredzētas,vai konfigurētas vietas, Ievāktie dati tiek apkopoti un ievietoti attiecīgajā datu glabāšanas struktūrā.
4. Informācija par pārbaudāmo pirmkodu tiek nogādāta lexer.bin.

5. Tiek sanemta informācija par doto pirmkodu.
6. Informācija tiek pārveidota uz datubāzē glabājamu tipu. Pēc LEX\YACC likumiem pirmkods tiek pārveidots par programmai saprotamu datu kopumu.
7. Dati tiek ievietoti datubāzē un pārveidoti ar specifisku algoritmu, kura rezultātā padotā pirmkoda mainīgo kārtība ir sakārtota noteiktā secībā. Mainīgo vārdi ir standartizēti un tiek izveidotas bloku salīdzināšnas vajdzībām iekodētas datu virknes. Šis solis nākotnē palīdz salīdzināšanai starp diviem pirmkoda piemēriem.
8. Ievākto datu salīdzināšana.
9. Tiek izveidoti visi nepieciešamie organismi. Katrs organisms pārbauda vai pirmkods ir kompilējams un paziņo par izpildes rezultātiem Pasaulei.
10. Pasaule salīdzina pirmkoda rezultātus.
11. Balstoties uz visu ievākto informāciju tiek izlikts vērtējums par datu līdzību.

3. MODUĻI

Šī nodaļa vairāk fokusējās uz katru moduli atsevišķi un izstāsta tā mērķus, iespējas, moduļa nepieciešamību. Iekšējās datu plūsmas un metožu izejas un izvades dati.

3.1. Datu iegūšana no pirmkoda

Visi dati sākumā tiek ieguti no padota C valodas pirmkoda. Šis pirmkods tiek pārveidots par atslēgvārdiem, kuri vēlāk grupās veido loģisko vienumu, piemēram, *int a = 0;* jebkurš kaut cik zinošs programmētājs saprot ka tika definēts veselas vērtības tipa mainīgais 'a' ar sākotnējo vērtību , kura ir vienāda ar 0. Ar definēto atslēgvārdu palīdzibu programmas algoritms mēģina atpazīt tam padoto simbolu virkni, kā programmas kodu.

3.2. LEX

LEX ir datora programma, kuru izmantojot var būvēt leksikas analizatorus. Bieži tiek izmantota kopā ar YACC. Sākumā tiek uzrakstīti likumi kā veidot atslēgvārdus, ar LEX - vai dotajā gadijumā FLEX, kas ir Ubuntu Linux operētājsistēma LEX programmas implementācija - tiek uzģenerēts C programmēšanas valodā yy.lex.h datne.[4]

3.2.1. LEX atslēgvārdi

1. Pielikumā dotā tabula apraksta uz doto brīdi eksistējošos atslēgvārdus, kuri tiek izmantoti, lai izveidotu leksikas analizatoru. 1. Tabula apraksta iepriekš definētas maskas, piemēram, decimāli un heksadecimāli skaitļi, vārds, atstarpe, jaunas rindas simbols u.c. 2. Tabula satur definēto masku kopumu un konkrētu elementu grupas, kuras izveido us. Šie atslēgvārdi tiek izmantoti BNF likumos YACC programmas daļā, lai izveidotu paziņojumu world.bin.

3.3. LEX/YACC mijiedarbība

Kā jau minēts LEX tiek bieži izmantots ar YACC. Ja LEX definēt leksiku un pārveido padoto tekstu par atslēgvārdu virkni, tad YACC pārveido atslēgvārdus sakārtotus BNF[6] par autora izveidotaj programmaturai sparotamu datu virkni.

Šīs abas lietotnes rada spēcīgu rīku jebkādu datu apstrādei. Tieši tāpēc tie tika izvēlēti ar

nolūku “tulkot” C programmēšanas valodu. Tālāk tiek aprakstīts kādā veida dati tiek padoti un saņemti atpakaļ no dotā moduļa.

Līdz ko *lexer.bin* ir pabeidzis inicializāciju. Tas pieprasa datus no pasaules par testējamo datņu daudzumu. Saņemot šo informāciju katrs no saņemtajām datnēm tiks izlaists cauri tālāk aprasktītai procedūrai

Sākumā LEX pārveido visus ieejas datus no saņemtās datnes par atslēgvārdu virkni. Vēlāk tiek izsaukta YACC procedūra, kura izmantojot BNF[6] tipa likumus mēģina atpazīt kādu vajadzigu informāciju. Tai brīdī, kad tiek sasniegts stāvoklis, kurā var viennozīmīgi pateikt, ka pēdējo atslēgvārdu kopums nes sev līdzi kādu jēgu, tas tiek ielikts ziņojumā priekš pasaules. Pēc ielikšanas iešana cauri atslēgvārdiem turpinās tādā pašā principa līdz tiek sasniegtas pārveidotu ievaddatu beigas.

Dati tiek apkopoti datu ziņojuma struktūrā tiks aizsūtīti pasaulei uz apstrādi. Šis pats process ar apstrādi tiks uzsākts ar nākošo pārbaudes datni. Tāds cikls turpinās līdz visas datnes tika pārtulkotas datu struktūras un aizsūtīti world.bin. Pēc šī brīža lexer.bin paziņo pasaulei, ka tas esot pabeidzis savu darbu un izslēdzas. Ja world.bin vajdzēs asptrādāt kādu citu datni, tad tā zinās ka vajag atkal sākt visu processu no 0 punkta.

3.4. Komunikātors / “Communicator”

Komunikācijai paredzēts modulis. Zem šī moduļa ir apkopota visa komunikācija starp dažādām binārām datnēm. Komunikācija notiek ar ligzdu palīdzību. Visa saziņa notiek ar iepriekš definētiem ziņojumiem.

3.4.1. Ziņojumu veids

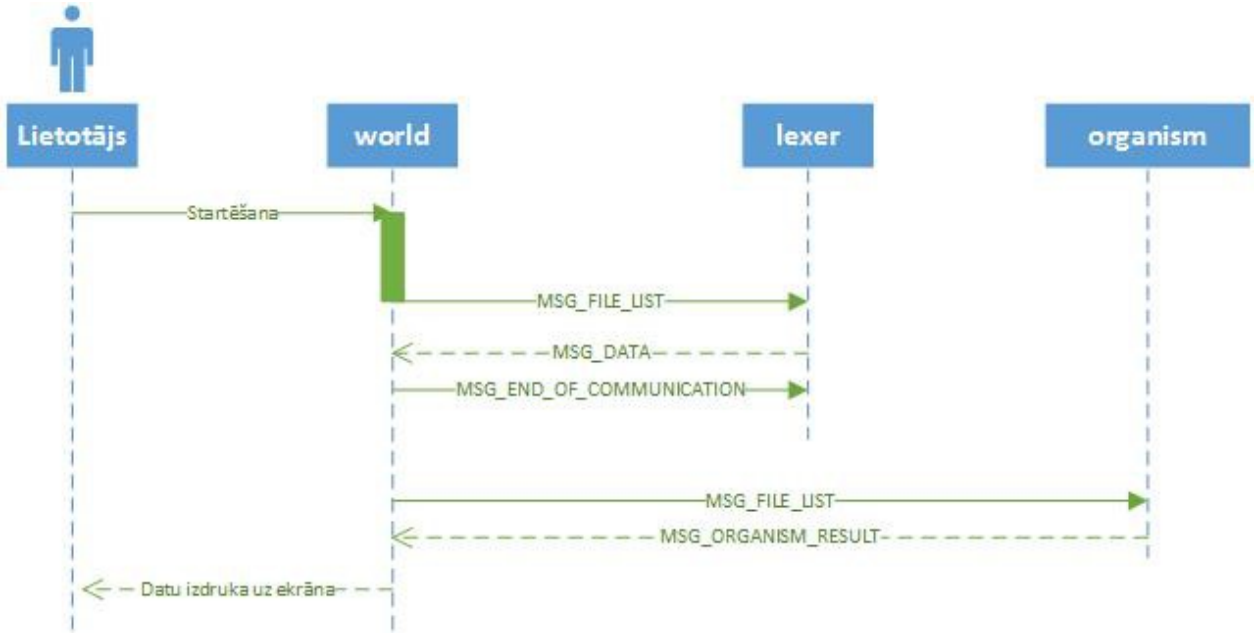
Kopā eksistē četri ziņojumu veidi. Pirmais ir tukš ziņojums, kurš palīdz noteikt to, ka savienojums joprojām eksistē. Tas ir nepeiciešams, ja savienojums tiks pārtraukts, tad pēc 5 sekundēm tas tiks pilnībā iznīcināts. Otrais ir testējamo datņu saraksts, kurš tiek nosūtīs no world.bin uz lexer.bin. Uz doto brīdi šis ir vienīgais veids, kā lexer.bin var uzzināt par testējamo datņu eksistenci. Trešais ziņojums nāk no lexer.bin puses uz world.bin. Tas satur datus par tikko nolasīto pirmkoda datnēm. Šāds ziņojums sastāv no vairākiem ierakstiem. Pēdējais ziņojuma veids ir paredzēts komunikācijas pārtraukšanai. Ja tāds tiek nosūtīts, tad komunikācija tiek pārtraukta nskatoties uz visiem pārējiem procesiem.

3.4.2. Ziņojumu struktūra

Savienojuma eksistences un komunikācijas pabeigšanas ziņojumi nesatur nekādus papildus datus. Testējamo datņu saraksts un nolasīto datu struktūru saraksts ir sadalīti ierakstos. Katrs ieraksts var satūrēt atslēgu, vārdu un vērtību. Datnes saraksta gadijumā atslēgas un vērtības lauki netiek izmantoti. Kad tiek sūtīti dati par pirmkoda datņu atslēgas lauks satur ieraksta tipu, vārda lauks satur mainīgā tiešo nosaukumu un vērtības laukā var atrasties vērtība. Vērtības un vārda lauki ne vienmēr tiek izmantoti. Šo datu pārveidošana par datubāzes ierakstiem sīkāk ir paskaidrota punktā 3.4 Pārveidotājs.

3.4.3. Tipiskas kumunikācijas piemēri

Komunikācija notiek starp visiem binārām datnēm. 3.1.Attēls parāda komunikāciju starp world.bin, lexer.bin un organism.org, nekādiem citiem, papildus, ziņojumiem nevajadzētu eksistēt starp dotiem moduļu kopumiem.



3.1. att **Ziņojumu secība**

3.5. Pārveidotājs / “Transformer”

Pārveidotājs ir pirmais pasaules modulis, kurš apstrādā datus. Šī moduļa galvenais uzdevums ir pārveidot infromāciju no lexer.bin par saprotamām datu struktūrām (skatīt 3.5.2 Nukleotīdu Struktūra), un kopā ar Primal datu bāzes palīdzibu tos saglabāt AVL kokā.

Datu pārveidošanai tiek izmantotas vairākas metodes, galvenās divas būtu *transform* un *createNucleotide*. *Transform* metode pārveido atsūtītos datus no lexer.bin par datubāzei saprotamiem. Katrs atsūtītā ziņojuma ieraksts tiek padots metodei *createNucleotide.* Šī metode padotos datus – tris simboliskas vriknes – pārveidos par nukleotīdu. Rezultāts šīm darbībām ir pamata datu nokļūšana datubāzē.

Pārveidošana no 3 simboliskām virknēm – tips, vārds, vērtība – par nucleotide_t mainīgo notiek sekojšā veidā:

1. Iegūstam atmiņu no sistēmas, lai tajā izvietotu jaunos datus.
2. No tipa virknes tiek iegūts jaunizveidota nukleotīda tips un apakštips.
3. Balstoties uz tipu un apakštipu tiek pārveidota vērtības virkne.
4. Izveidotie dati tiek iekodēti ar metodi createNucleobase, šie dati tiks izmantoti salīdzināšanai.
5. Ja nebija nekādu kļūdu, tad nucleotīds tiek piesaistīts pie nukleotīdu kopējās struktūras.
6. Pēc *createNucleotide* veiksmīgas pabeigšanas dati tiek pievienoti AVL datubāzei.

Šis process ir ļoti svarigs, jo visas pārējās darbības notiek jau ar datubāzes datiem un sākotnējais ziņojums par datu struktūra tiks dzēsts. Ja šī soļa laikā tiks ielaista sasaistes kļūda, tas var rast programmai citu priekšstatu par pārbaudāmo pirmkodu.

3.6. Datubāze / “Primal”

Šī ir datu bāze, kura sevī uztur visus atrastos nukleotīdus. Galvenie dati, protams, ir AVL[7] koka instancē. AVL koka dati gan nav saistiti tikai ar AVL koka iespējam, bet tiem paralēli eksistē atsevišķa datu sasaiste – tiek izveidotas ieejas datu nukleotīdu ķēdes.

3.6.1. Nukleotīdu struktūra

Nukleotīdi sastāv no vairākiem datiem, daļa no informācijas ir obligāta, citi lauki tiks izmantoti tikai noteikta veida nukleotīdiem. Pati struktūra ir atrodama 2. Pielikumā. Dati lauki ir iedalīti divās daļās: kopīgie visiem nukleotīdu tipiem un specifiskie katram nukleotidu apakštipam.

File un *name* lauki ir diezgan pašsaprotami. *File* apraksta nukleotīda piederību kādai (vai kādām datnēm. Name ir tiešais mainīgā nosaukums. *Nucleobase* nosaka nukleotīda tipu un apakštipu, tips var būt viens no:

1. NUCLEO_TYPE_BASE;
2. NUCLEO_TYPE_CONTROL;
3. NUCLEO_TYPE_LOOP;
4. NUCLEO_TYPE_JUMP;
5. NUCLEO_TYPE_SUPPORT;
6. NUCLEO_TYPE_ASSIGNS;
7. NUCLEO_TYPE_COMPARE;
8. NUCLEO_TYPE_OPERATOR.

Tieši šis lauks pasaka tiešo nukleotīda tipu. Šo tipu dažādās iespējas ir aprakstītas 2. Pielikumā. *Parent* un *sibling* ir norādes uz attiecīgi, augstāk stāvošu nukleotīdu un uz nākošo izpildes secībā esošu nukleotīdu.

Salīdzināšanas nolūkos tiek izmantots speciāls lauks - nucleobase, kurš reprezentē nukleotīda saturu. Gadijumā, ja tas ir kāds nukleotīds, kurš var aptvert vairākus nukleotīdus kopā, piemēram, “if”, “for”, funkcijas utl, tad šis lauks apzīmē ne tikai dotā nukleotīda loģisko saturu, bet arī tai ietverto nukleotīdu kopējo nozīmi.

Atgādinam, ka *nucleobase* lauks tiek veidots dautu pievienošanas laikā, kad tiek izveidots

pats nukleotīds. Ja dotais nukleotīds tiek pievienots kādam augstāk stāvošam, tātad aptverošam, nukleotīdam, tad tas, kuram tiek pievienots, tiek arī papildināts ar jauno informāciju. Šī informācija tiek glabāta specifiskā veidā.

Lauks *nucleobase_count* apraksta cik daudz nukleotīdu ir aprakstīts nucleobase laukā. Pats nukleobase lauks ir norāde uz atmiņu, kur glabājās konkrēta nukleotīda dati. Ir izveidota speciala tabula, kura apraksta kāds tips tiks apzīmēts ar kādu simbolu.

Nucleobase tiek veidots pēc sekojoša principa. Katrs nucleobase ieraksts ir 64bitus garš. Pirmie trīs biti apzīmē tipu, nākošie pieci ir apakštips. Tas ir pirmais oktets. Nākošie 56 biti ir numurs pēc kārtas savam vecākam. Gadijumā, kad dotajam nukleotīdam ir bērni. Tad dotā nukleotīda nucleobase satur arī pēc izsaukuma sakārtotus ierakstus, kuri ir bērnu nucleobase ieraksti. Gadijumā ja dotajam nukleotīdam nevar būt neviena bērna, tad eksistēs tikai viens ieraksts un *nucleobase_count* būs vienāds ar viens.

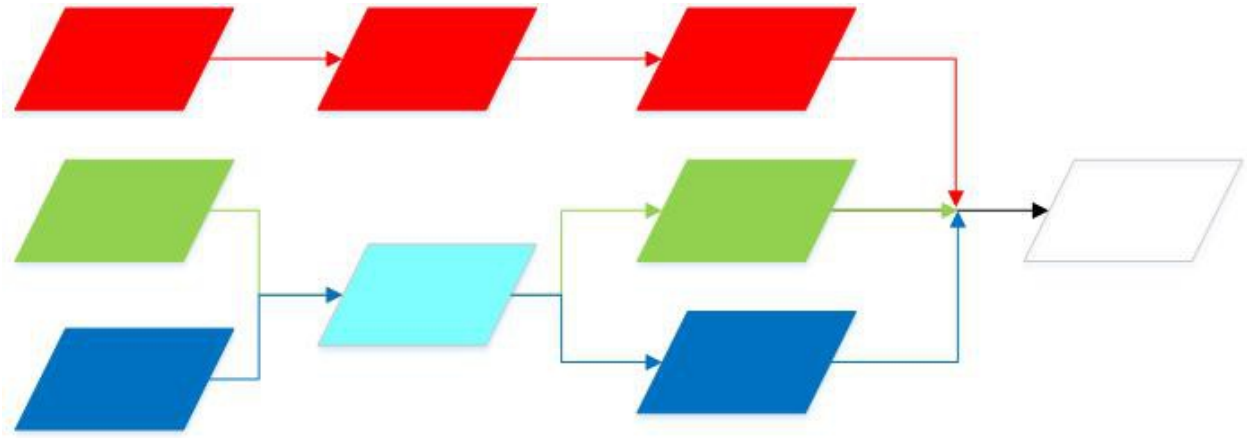
Autors izvēlējas šādu pieeju, pretstatā kopējā ceļa līdz dotajam nukleotīdam glabāšanas paņēmienam, jo tādā gadijumā tiktu iznīcināta iespēja vairākkārtēji izmantot vienu un to pašu nukleotīdu, vai to kopumu. Nebūtu datu ar kuriem varētu identificēt specifisko datu kopumu un dati par ceļu nebūtu viennozīmīgi visiem nukleotīdiem.

Jāpiemin ka *nucleobase* dati tiek nedaudz pārkārtoti. Tai brīdī, kad notiek datnes satura kārtošana.

Ērtībai ir izveidotas palīgmetodes, kuras pievieno, atjauno vai dzēš šos datus. Pievinošanas brīdī var gadīties tāda situācija, ka iepriekšēji izdalītais atmiņas apjoms ir nepietiekams, un papildus bloks ir pieprasīts.

3.6.2. Datu glabāšanas veids

Par AVL koku struktūru un algoritmiem var izlasīt rakstā "Algoritms par informācijas organizēšanu"[7] Īsumā tas ir binārs koks ar īpašu zaru balansēšanas algoritmu. Paralēli jau eksistējošām ierakstu saitēm eksistē nukleotīdu ķēdes. Šādas ķēdes var saturēt kopīgus elementus, lai divi vienādi nukleotīdi netiktu uzturēti vienā datubāzē.



3.2. att **Nukleotīdu bloku ķēde**

Attēlā katrs paralelograms reprezentē nukleotīdu. Ir trīs ķēdes (sarkana, zāļa un zila). Kā redzams zaļā un zilā ķēde dala kopīgu otro elementu un visām trim ķēdēm ir kopīgs elements. Piemēram, lielākai daļai C programmu main funkcija beidzas ar "*return 0;*".

Katrs nukleotīds satur divas atsauces – viena uz savu vecāku, otra uz sekojošo elementu. Bez šīm atsaucēm, gadijumā ja dotais nukleotīds

3.6.3. Datu meklēšana

AVL koku bibleotēka, kura tiek izmantota darba izstrādē, piedāvā eksistējošu meklēšanas iespēju. Tai tikai ir jāizveido salīdzināšanas funkcija. Šī funkcija tiek izmantota arī koka balansēšanas laikā. Salīdzināšana notiek izmantojot nukleotīda *nucleobase*. Pati salīdzināšana notiek izmantojot memcmp funkcijas izsaukumu ar argumentiem, kuri ir divu nukleotīdu *nucleobase* rādītāji. Tādā veidā lauku un mainīgo vārdi neietkmē loģiskā risinājuma būtību, tas savukārt palīdz atklāt divus vienlīdzīgus algoritmus.

3.7. Analizētājs / “Analyser”

Šī moduļa nolūks ir sakārtot datus un vēlāk izveidot statistisko analīzi par tiem. Šīs procedūras arī to dara tādā secībā. Sākumā sakārto datus iekš katra nukleotīda bloka, paralēli salabojot *nucleobase* atsauces augstāk, jo pārkārtošana ietekmē arī visu augstākstāvošo nukleotīdu datus.

3.7.1. Pārkārtošanas prioritātes

Visi tālāk minētie likumi tiek pielietoti kārtojot iekš katra atsevišķa tvēruma. Augstāk un zemāk eksistējošie nukleotīdi tiek kārtoti pirms vai pēc. Šī metode ir rekursīvi izsaucama uz katru apakšelementu.

1 Mainīgo definējuma nukleotīdi tiek pārvietoti uz tvērina sākumu.

2 Mainīgo definējumi tiek sakārtoti pēc to izmēriem atmiņā. Sakrišanas gadijumā:

  2.1 Tiek atrasts pirmais izmantošanas brīdis, kurš ietkmē jebko citu izņemot pašu mainīgo, kas ļauj definēt kurš mainīgais ir kurš.

  2.2 Atrastas atšķirīgas darbības ar diviem salīdzinājumiem, tad tiek izmantoti matemātiskie likumi (reizināšana pirms summēšanas utl.) prioritātes noteikšanai.

3 Pārējās darbības, metožu un funkciju izsaukumi paliek savā izpeldes secībā.

3.7.2. Salīdzināšana

Kā jau minēts iepriekš lauks nucleobase tiek izmantots, lai atrastu vienādus nukleotīdu blokus, kuri ir izmantoti dažādās datnēs. Faktīski ir jāuztaisa pilnās pārlases meklēšanas ar salīdzinājumiem pēc garuma. Jo jebkura datne var saturēt jebkurā citā datnē realizētus bloku.

Process notiek salīdzinot A un B nukleotīdu blokus. Bloka A nucleobase_count ir salīdzināts ar B bloka nukleotīda nukleobase_count. Datņu salīdzināšanas gadijumā tiek ignorēti pašu datņu nucleobase ieraksti (pirmā pozīzcijā esošie). Ja izmēra ziņā bloks A var potenciāli ietilpt blokā B, tiek veikta salīdzināšana ar bloka A garumu un blokā B nobīdot rādītāju pa vienu pozīciju tik ilgi,kamēr bloka B atlikušā garumā vēl var ietilpt bloks A. Atomārie dati netiek meklēti citos blokos, tādi kā mainīgo definīcijas, īsās darbības.

3.7.3. Analīzes rezultāti un potenciāls

Vienkāršākais ko var iegūt ir bloku popularitāte, jeb cik bieži viens un tas pats bloks tiek izmantots dažādās datnēs. Tā kā lielākā mērogā tās varētu būt veselu algoritmu realizācijas. No tā varētu vērtēt uzprogrammēšanas sarežģītību konkrētam algoritmam. Tādas informācijas iegūšanai, protams, arī ir vajadzīga lielāka auditorija, jeb testa datnes.

Papildus bloku grupēšana un popularitāte var uzrādīt līdzīgus vai identiskus risinājumus, kuri varētu būt uzskatīti par plaģiātu. Vismaz tiktu izlikts brīdinājums par tādu varbūtību, kas likt cilvēkam pārbaudīt abu pirmkoda eksemplārus.

Sekojoši var arī vilkt paralēles starp izmantotiem algoritmiem un pirmkoda autoru spējām. Iespējams arī analizēt sociālos aspektus – cik bieži konkrēto autoru darbi ir savstarpēji tuvuiem rezultātiem. Studiju vidē varētu sasaistīt ar autora vidējiem vērtējumiem, vērtējumiem konkrētā tēmā.

Kopumā datu izmantošana ir lietotāja paša izvēle. Plānots pievienot iespēju izvadīt datus kādā plašāk pieejamā datu formātā, nevis tikai izdrukas uz ekrāna. Tas ļautu veikt daudz sekmīgāku intergrēšanu ar citām sistēmām.

3.8. Organisms / “Organism”

Dotais modulis tiek izmantots, lai pārbaudītu atsevišķa pirmkoda darbību. Šī moduļa galvenais mērķis ir mēģināt nokompilēt un vēlāk pārbaudīt vai dotā pirmkoda izveidotā datne spēj veiksmīgi funkcionēt.

Modulis ir atsevišks binārs fails, kurš tā pat, kā lexer.bin pieslēdzās pie world.bin un iegūst sarakstuu ar datnēm. Tās tiek nokompilētas. Ja viss notiek sekmīgi, tad tiks iegūti dati par izvada rezultātiem. Šie rezultāti ietekmēs programmas sekmības vērtējumu. Uz doto brīdi Organisms, izņemot kā ar kompilēšanu un programmas izpildi ne ar ko citu nenodarbojas.

4. DATU APSTRĀDES CEĻŠ

Šī nodaļa apskata datu pilnu ceļu caur visu izveidoto programmatūru. Atšķirība no informācijas, kura ir minēta pie moduļu apraksta, galveno kārt tiek fokusēts tieši dati un viņu transformācijas katrā modulī.

Kāds ar programmatūru nesaistīts cilvēks izveido savu programmu. Piemēra pēc tiks izmantots tipiskākā iesācēju programma, kuru izmanto jebkurā valodā, bet dotā gadijumā C valodas “Hello World” piemērs.

```
#include <stdio.h>

int main(){
    printf("Hello World!\n");
    return 0;
}
```

Automatizēts servers saņems šādu datni un izvietos to kādā no iepriekš definētām mapēm. Pēc pārstrādes tikai ar LEX tiek iegūta šāda virkne:

INT IDENTIFIER '(' ')' '{' IDENTIFIER '(' STRING_LITERAL ')' ';' RETURN I_CONSTANT ';' '}'

Izmantojot 1. Pielikuma tabulas to var izdarīt arī bez programmas starpniecības. Tomēr LEX un YACC kopējās darbības laikā process nav tik tiešs, kā padot datus LEX, tad tos pārraidīt uz YACC un viss strādā.

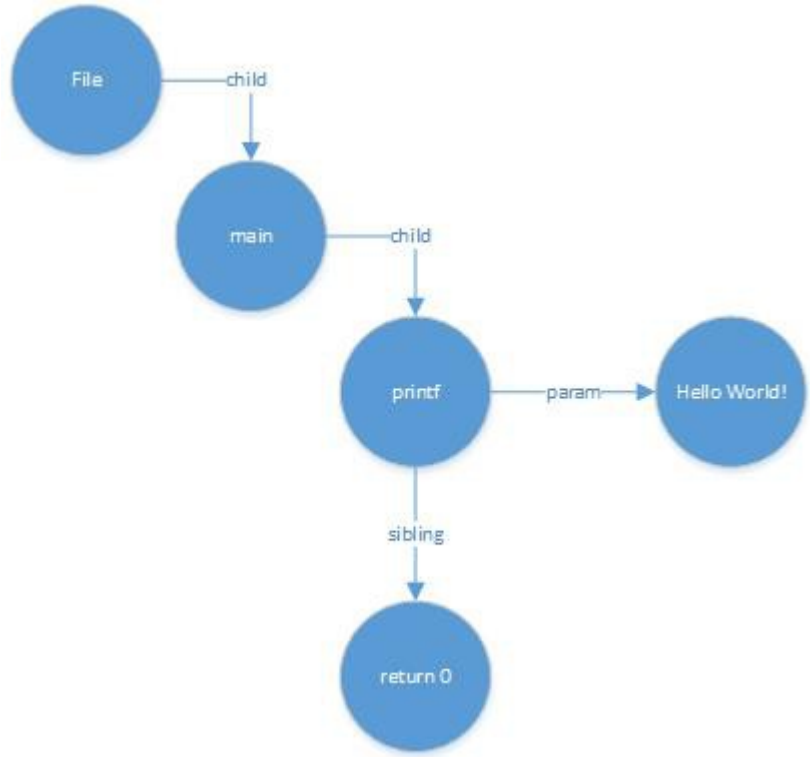
Pamata lietotšanas gadijumā LEX un YACC sākumā tiek izveidoti specialas datnes lex.yy.c, y.tab.c un y.tab.h,[4][5] kā arī ir jāpiesaista libl kompilēšanas laikā. Tātad LEX un YACC netiek nolikti secīgā izpildē, drīzāk tie tiek sakausēti kopā vienā spēcīgā programmatūras kopumā. Attiecīgi tā virkne kura tika parādīta iepriekš īstenībā neatbilst gluži tam ko redz YACC puse. Bet tie likumi tiek joprojām izmantoti. YACC vadoties ar visiem izveidotiem likumiem pakāpeniski savāc datus un saglabā tos kopējā paziņojumā. Kurš, izdrukājot uz ekrāna, izskatās aptuveni šādi:

```
 int | main |
block_st |  | printf
fnc_srt | printf |
args_str |  | "Hello World!\n"
string |  | "Hello World!\n"
args_end |  | )
return | 0 | 0
block_en |  | }
```

Šādā veidā Transformer modulis arī saņem datus no lexer.bin. Izņemot to ka ir arī galvene, kurā ir norādīts cik ierakstu seko viens otram. Transformer modulis attiecīgi sāk iet cauri katram ziņojuma ierakstam un pārveidot tos par nukelotīdiem.

```
Name                    Type            Subtype
./tasks/hello.c         SUPPORT                 FILE START
8900000000000000 2000000000000000 8200000000000001 a00000000000001 6000000000000002
main                    CONTROL                 FUNCTION
2000000000000000 8200000000000001 a00000000000001 6000000000000002
printf                  CONTROL                 FUNC_SRT
8200000000000001  a00000000000001
“Hello World”           BASE            STRING
a00000000000001
return 0                JUMP            RETURN
6000000000000002
```

Pēc transformācijas procedūras tiek iegūta augstāk redzama izdruka (jāpiezīmē, ka izdruka ir nedaudz ar plašāku informāciju un tās veids var atšķirties balstoties uz izstrādes gaitu), Nav grūti vilkt paralēles ar ieejas datiem.



4.1. att **Nukleotīdu struktūra**

4.1. Attēls attēlo saites starp izveidotiem nukleotīdiem, ne visas saites ir attēlotas. Attēlotie dati pirms datu pārveidošanas bija tikai kā teksta kopums, pretstatā šādam struktūru kopumam, kuru var turpmāk izmantot un iegūt lietotājam vajadzīgo informāciju. Nākošais solis ir analizātors. Tas saņem datus par visām datnēm kopā. Pirmais, kas tiek izdarīts ir datu pārkārtošana. Diemžēl ar tik primitīvu piemēru kā ir dots nekādu pārkārtošanu nenotiks. Pārkārtošanas princips tika aprkastīts arī 3.7.1 nodaļā

Analizātors datus datubāzē nepārveido. Tas tikai izvada statistikas datus uz erkāna vai iepriekš konfigurācijā uzstādītā datnē. Datu izvade gan atstāj vēlēties ko vairāk. Nākotnē būs realizētas labākas metodes. Šis ir ļoti svarīgs aspekts programmatūras pielietošanai, savādāk to ir ļoti grūti savienot ar citiem rīkiem.

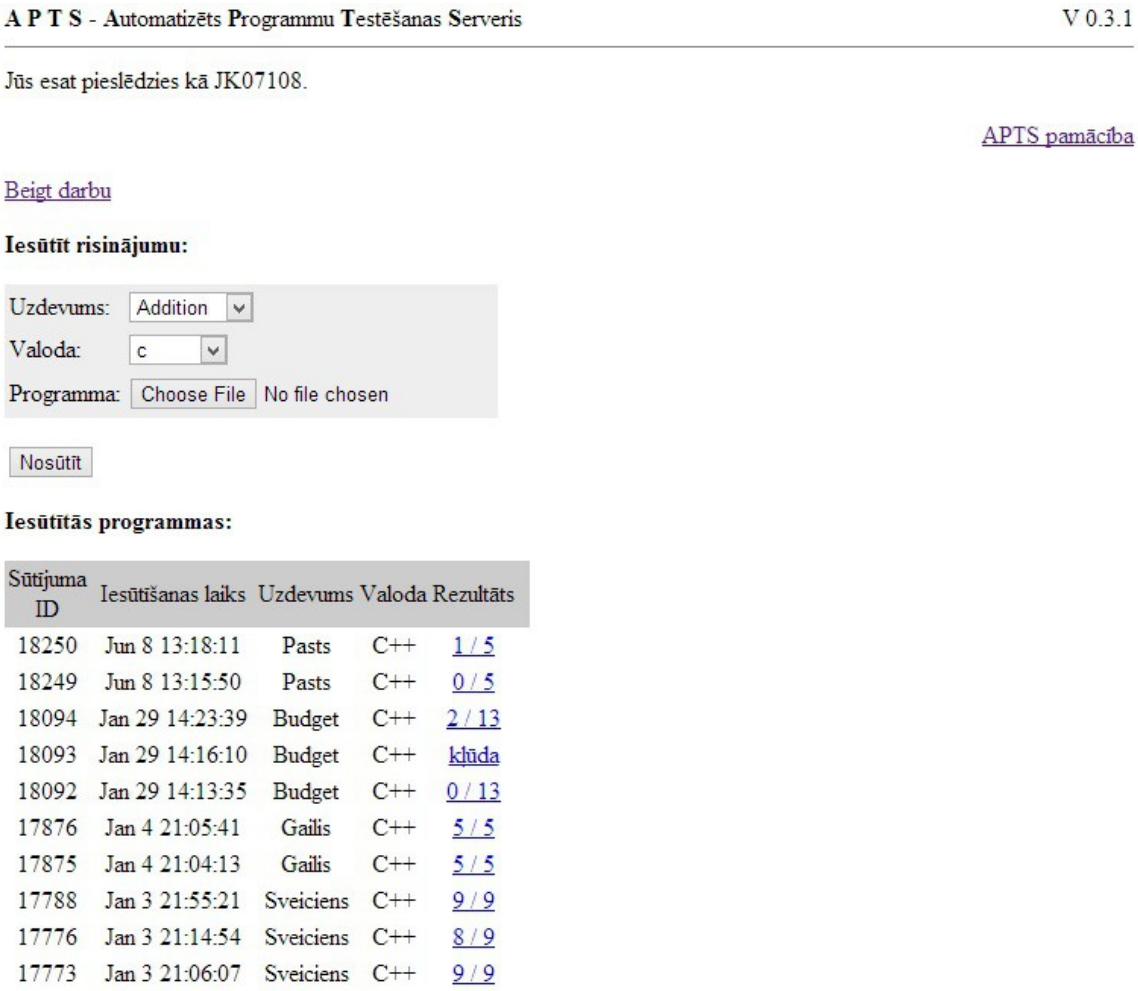
Dotajā piemērā gan nav nekādu datu ko izvadīt, jo nebija otras datnes ar kuru salīdzināt. Kā arī lai iegūt pēc iespējas pilnākus datus būtu nepieciešami gan sarežģītāki, gan vairāk ieejas datu. Diemžēl izstrādes gaitā netika izvēlēta šāda testētāju grupa. Tas daļēji ir arī sakrā ar to, ka programmatūra joprojam atrodas izstrādes procesā un uz doto brīdi spēj atpazīt tikai dizegan vienkāršas pirmkoda struktūras.

5. SALĪDZINĀJUMS AR LĪDZĪGU PORGRAMMATŪRU

Dotās programmatūras mērķīs ir iegūt datus par vairākām, līdzīgām pirmkoda datnēm. Ievāktā informācija vairāk ir attiecināma uz izmantoto algoritmu un to realizācijām. Tālāk ir minēti daži rīki kuri nodarbojās ar ko līdzīgu.

5.1. APTS

Lielākai daļai Latvijas Universitātes Datorikas fakultātes studentiem ir pazīstams šis saīsinājums, kurš nozīmē Automatizēts Programmu Testēšanas Serveris. Girta Folkmaņa un Gunta Arnicāna izstrādāts rīks, kurš kā ieejas datus saņem studentu veidotus risinājumus un rezultātā izveido statistiku par studentu algoritma realizāciju sekmēm.



A P T S - Automatizēts Programmu Testēšanas Serveris V 0.3.1

Jūs esat pieslēdzies kā JK07108.

APTS pamācība

Beigt darbu

Iesūtīt risinājumu:

Uzdevums: Addition

Valoda: c

Programma: Choose File No file chosen

Nosūtīt

Iesūtītās programmas:

Sūtījuma ID	Iesūtīšanas laiks	Uzdevums	Valoda	Rezultāts
18250	Jun 8 13:18:11	Pasts	C++	1 / 5
18249	Jun 8 13:15:50	Pasts	C++	0 / 5
18094	Jan 29 14:23:39	Budget	C++	2 / 13
18093	Jan 29 14:16:10	Budget	C++	kļūda
18092	Jan 29 14:13:35	Budget	C++	0 / 13
17876	Jan 4 21:05:41	Gailis	C++	5 / 5
17875	Jan 4 21:04:13	Gailis	C++	5 / 5
17788	Jan 3 21:55:21	Sveiciens	C++	9 / 9
17776	Jan 3 21:14:54	Sveiciens	C++	8 / 9
17773	Jan 3 21:06:07	Sveiciens	C++	9 / 9

5.1. att **APTS**

APTS izmanto publisko un slēpto testu kopumus, lai noteiktu cik sekmīgs bija risinājums. Šis rīks (cik zināms) gan neveic iekšējo pirmkoda analīzi, notiek tikai rezultātu un izejas kodu pārbaudes.

5.2. Cloc un līdzīgi

Cloc[8] ievāc datus par izmantotām programmēšanas valodām, pirmkoda rindiņu skaits tajās, komentāru daudzums. Šāda tipa rīki ir arī Sonar, Ohcount, SLOCCount, sclc, Codecount un loc. Attēlā var redzēt cloc izvadu, kur tiek analizēts pirmkods izstrādātaj programmatūrai.

```
janisknets@ubuntu:~/Projects/bakdarbs$ cloc ~/Projects/bakdarbs/
      46 text files.
      46 unique files.
    1015 files ignored.

http://cloc.sourceforge.net v 1.56  T=0.5 s (72.0 files/s, 8702.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
C++                             13            173             84           1492
yacc                             2            192              0            972
C/C++ Header                    12            103            145            586
lex                              2             38              2            352
make                             1             35             12            113
C                                4              3              0             26
HTML                             1              1              0             11
XML                              1              0              0             11
-------------------------------------------------------------------------------
SUM:                            36            545            243           3563
-------------------------------------------------------------------------------
janisknets@ubuntu:~/Projects/bakdarbs$ date
Mon Jun  3 01:40:44 EEST 2013
janisknets@ubuntu:~/Projects/bakdarbs$ _
```

5.2. att: **Cloc**

Eksistē arī citi rīki, tomēr netika atrasts neviens, brīvi pieejams rīks, kurš nodarbotos ar datņu iekšējās uzbūves un algoritmu izpratni.

6. REZULTĀTI.

Veiksmīgi izdevās izveidot pamata moduļus, kuri spēj atpazīt pirmkodu. Atpazīšana un pārveidošana par programmai saprotamu struktūru vajadzētu uzlabot. Kā arī datu izvades formu vajadzētu pilnveidot. Uz doto brīdi rīks veic tikai dažas pamata funkcijas, bet ar to arī viss aprobežojās.

Neskatoties uz nepilnībām dotajā brīdī, programmatūra netiks atstāta nepabeigta. Dotā uzdevuma kontekstā tiks noteikti pilnveidota, kā arī iegūs papildus funkcionalitāti, kura ir saistīta ar ievāktās informācijas pārveidošanu par jaunu pirmkodu. Tā kā pirmkoda analīze ir viena no pamata nepieciešamībām, bez kuras nav iespējams virzītos tālāk. Analīzej ir jābūt nevainojamai, un daudzpusīgai.

Programmatūras nepabeigts stāvoklis ir saistīts ar autora pieredzes trūkumu darbojoties ar LEX\YACC programmatūru, likumu izstrāde, pat izmantojot ar pamatu jau uzrakstītus likumus, aizņēma pārāk daudz laika. Netika pareizi novērtēta darbietilpība un grūtības pakāpe. Tomēr šīs zināšanas ir ļoti vērtīgas un noteikti tiks izmantotas nākotnē.

7. IZMANTOTĀ LITERATŪRA

1. **Knets J.** *Pamatmodelis Moduļu-bāzētai Mākslīgai Dzīvībai,* 2012
2. **Harman M.** *Why Source Code Analysis and Manipulation Will Always Be Important.* 2010 Pieejams: http://www0.cs.ucl.ac.uk/staff/mharman/scam10.pdf
3. **Niemann T.** *Lex and Yacc Tutorial.* Pieejams: http://epaperpress.com/lexandyacc/
4. **Degener J.** *ANSI C Yacc grammar.* 1995 Pieejams: http://www.lysator.liu.se/c/ANSI-C-grammar-y.html
5. **Degener J.** *ANSI C grammar, Lex specification.*1995 Pieejams: http://www.lysator.liu.se/c/ANSI-C-grammar-l.html
6. **Backus J.W.** *The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference*. 1959 Pieejams: http://www.softwarepreservation.org/projects/ALGOL/paper/Backus-Syntax_and_Semantics_of_Proposed_IAL.pdf
7. **Г. М. Адельсон-Вельский, Е. М. Ландис.** *Один алгоритм организации информации* // Доклады АН СССР. 1962. Т. 146, № 2. С. 263–266.
8. **Danial A.** *CLOC* [Tikai tiešsaistē] Pieejams: http://cloc.sourceforge.net/
9. **Jānis K.** *Pirmkoda statistiskās analīzes automatizācijas github projekts* [Tikai tiešaistē] Pieejams: https://github.com/taureliloome/bakalaurs

1. pielikums.

LEX definīciju tabulas

1. tabula

Atslēgvārds	**Maska**
O	[0-7]
D	[0-9]
NZ	[1-9]
L	[a-zA-Z_]
A	[a-zA-Z_0-9]
H	[a-fA-F0-9]
HP	(0[xX])
E	([Ee][+-]?{D}+)
P	([Pp][+-]?{D}+)
FS	(f|F|l|L)
IS	(((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?))
CP	(u|U|L)
SP	(u8|u|U|L)
ES	(\\(['"\?\\abfnrtv]|[0-7]{1,3}|x[a-fA-F0-9]+))
WS	[\t\v\n\f]

2. tabula

"/*"	comment
"//".*	/* consume //-comment */
"#include <"{A}*"."{A}*">"	/* ignore */
"#include \""{A}*"."{A}*"\""	/* ignore */
"auto"	AUTO
"break"	BREAK
"case"	CASE
"char"	CHAR
"int"	INT
"short"	SHORT
"long"	LONG
"void"	VOID
"signed"	SIGNED

Maska	**Atslēgvārds**
"unsigned"	UNSIGNED
"const"	CONST
"continue"	CONTINUE
"default"	DEFAULT
"do"	DO
"double"	DOUBLE
"else"	ELSE
"enum"	ENUM
"extern"	EXTERN
"float"	FLOAT
"for"	FOR
"goto"	GOTO
"if"	IF
"inline"	INLINE
"register"	REGISTER
"restrict"	RESTRICT
"return"	RETURN
"sizeof"	SIZEOF
"static"	STATIC
"struct"	STRUCT
"switch"	SWITCH
"typedef"	TYPEDEF
"union"	UNION
"volatile"	VOLATILE
"while"	WHILE
"_Alignas"	ALIGNAS
"_Alignof"	ALIGNOF
"_Atomic"	ATOMIC
"_Bool"	BOOL
"_Complex"	COMPLEX
"_Generic"	GENERIC
"_Imaginary"	IMAGINARY
"_Noreturn"	NORETURN
"_Static_assert"	STATIC_ASSERT
"_Thread_local"	THREAD_LOCAL
"__func__"	FUNC_NAME

Maska	**Atslēgvārds**
{L}{A}*	IDENTIFIER
{HP}{H}+{IS}?	I_CONSTANT
{NZ}{D}*{IS}?	I_CONSTANT
"0"{O}*{IS}?	I_CONSTANT
{CP}?"'"([^'\\\n]|{ES})+"'"	I_CONSTANT
{D}+{E}{FS}?	F_CONSTANT
{D}*"."{D}+{E}?{FS}?	F_CONSTANT
{D}+"."{E}?{FS}?	F_CONSTANT
{HP}{H}+{P}{FS}?	F_CONSTANT
{HP}{H}*"."{H}+{P}{FS}?	F_CONSTANT
{HP}{H}+"."{P}{FS}?	F_CONSTANT
({SP}?\"([^"\\\n]|{ES})*\"{WS}*)+	STRING_LITERAL
"..."	ELLIPSIS
">>="	RIGHT_ASSIGN
"<<="	LEFT_ASSIGN
"+="	ADD_ASSIGN
"-="	SUB_ASSIGN
"*="	MUL_ASSIGN
"/="	DIV_ASSIGN
"%="	MOD_ASSIGN
"&="	AND_ASSIGN
"^="	XOR_ASSIGN
"|="	OR_ASSIGN
">>"	RIGHT_OP
"<<"	LEFT_OP
"++"	INC_OP
"--"	DEC_OP
"->"	PTR_OP
"&&"	AND_OP
"||"	OR_OP
"<="	LE_OP
">="	GE_OP
"=="	EQ_OP
"!="	NE_OP
";"	';'
("{"|"<%")	'{'

Maska	**Atslēgvārds**
("}"|"%>")	'}'
","	','
":"	':'
"="	'='
"("	'('
")"	')'
("["|"<:")	'['
("]"|":>")	']'
"."	'.'
"&"	'&'
"!"	'!'
"~"	'~'
"-"	'-'
"+"	'+'
"*"	'*'
"/"	'/'
"%"	'%'
"<"	'<'
">"	'>'
"^"	'^'
"|"	'|'
"?"	'?'
{WS}	/* whitespace separates tokens */
.	/* discard bad characters */

Nukleotīda struktūra

```
typedef struct nucleotide_s {
    char file[FILE_NAME_MAX_LEN];
    char name[NUCLEOTIDE_NAME_MAX_LEN];
    nucleotide_s *parent;
    nucleotide_s *sibling;
    union {
        nucleotide_base_s base;
        struct {
            struct nucleotide_s *param_fst;
            struct nucleotide_s *param_lst;
            struct nucleotide_s *child_fst;
            struct nucleotide_s *child_lst;
        } control;
        struct {
            struct nucleotide_s *statement;
            struct nucleotide_s *child_fst;
            struct nucleotide_s *child_lst;
        } loop;
        struct {
            struct nucleotide_s *jump;
        } jump;
    } subvalues;

    uint64_t nucleobase_count;
    nucleobase_u *nucleobase;
} nucleotide_t;
```

3. pielikums.

Nukleotīda apakštipi

```
typedef enum {
    NUCLEO_BASE_VOID = 0,
    NUCLEO_BASE_CHAR,
    NUCLEO_BASE_SHORT,
    NUCLEO_BASE_INT,
    NUCLEO_BASE_LONG,
    NUCLEO_BASE_FLOAT,
    NUCLEO_BASE_DOUBLE,
    NUCLEO_BASE_SIGNED,
    NUCLEO_BASE_UNSIGNED,
    NUCLEO_BASE_BOOL,
    NUCLEO_BASE_STRING,
    NUCLEO_BASE_UNDEFINED
} nucleotide_base_e;
typedef enum {
    NUCLEO_CONTROL_FUNCTION = 0,
    NUCLEO_CONTROL_ENUM,
    NUCLEO_CONTROL_STRUCT,
    NUCLEO_CONTROL_IF,
    NUCLEO_CONTROL_ELIF,
    NUCLEO_CONTROL_ELSE,
    NUCLEO_CONTROL_SWITCH,
    NUCLEO_CONTROL_UNDEFINED
} nucleotide_control_e;
typedef enum {
    NUCLEO_LOOP_DO = 0,
    NUCLEO_LOOP_WHILE,
    NUCLEO_LOOP_FOR,
    NUCLEO_LOOP_UNDEFINED
} nucleotide_loop_e;
```

```
typedef enum {
    NUCLEO_JUMP_RETURN = 0,
    NUCLEO_JUMP_BREAK,
    NUCLEO_JUMP_CONTINUE,
    NUCLEO_JUMP_GOTO,
    NUCLEO_JUMP_UNDEFINED
} nucleotide_jump_e;

typedef enum {
    NUCLEO_SUPPORT_BLOCK_START = 0,
    NUCLEO_SUPPORT_BLOCK_END,
    NUCLEO_SUPPORT_FUNC_SRT,
    NUCLEO_SUPPORT_FUNC_END,
    NUCLEO_SUPPORT_FUNC_NAME,
    NUCLEO_SUPPORT_FUNC_PARAM,
    NUCLEO_SUPPORT_ARGS_START,
    NUCLEO_SUPPORT_ARGS_END,
    NUCLEO_SUPPORT_ARGUMENT,
    NUCLEO_SUPPORT_UNDEFINED
} nucleotide_support_e;

typedef enum {
    NUCLEO_ASSIGNS_IS = 0,
    NUCLEO_ASSIGNS_SUM,
    NUCLEO_ASSIGNS_MIN,
    NUCLEO_ASSIGNS_MULTIPLY,
    NUCLEO_ASSIGNS_DEVIDE,
    NUCLEO_ASSIGNS_MOD,
    NUCLEO_ASSIGNS_PLUS_ONE,
    NUCLEO_ASSIGNS_MINUS_ONE,
    NUCLEO_ASSIGNS_SHIFT_LEFT,
    NUCLEO_ASSIGNS_SHIFT_RIGHT,
    NUCLEO_ASSIGNS_AND,
    NUCLEO_ASSIGNS_OR,
    NUCLEO_ASSIGNS_XOR,
    NUCLEO_ASSIGNS_UNDEFINED
} nucleotide_assigns_e;
```

```
typedef enum {
    NUCLEO_COMPARE_EQUAL = 0,
    NUCLEO_COMPARE_NOT_EQ,
    NUCLEO_COMPARE_LESS,
    NUCLEO_COMPARE_MORE,
    NUCLEO_COMPARE_LESS_EQ,
    NUCLEO_COMPARE_MORE_EQ,
    NUCLEO_COMPARE_UNDEFINED
} nucleotide_compare_e;

typedef enum {
    NUCLEO_OPERATOR_PLUS = 0,
    NUCLEO_OPERATOR_MINUS,
    NUCLEO_OPERATOR_TIMES,
    NUCLEO_OPERATOR_DEVIDE,
    NUCLEO_OPERATOR_MOD,
    NUCLEO_OPERATOR_NOT,
    NUCLEO_OPERATOR_AND,
    NUCLEO_OPERATOR_OR,
    NUCLEO_OPERATOR_INVERT,
    NUCLEO_OPERATOR_PTR,
    NUCLEO_OPERATOR_UNDEFINED
} nucleotide_operator_e;
```

DOKUMENTĀRĀ LAPA

Bakalaura darbs "**Pirmkoda statistiskās analīzes automatizācija**" izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai

Autors: Jānis Knets

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: profesors Dr.sc.comp. Leo Seļāvo

Recenzents: asociētais profesors Dr.sc.comp. Jānis Zuters

Darbs iesniegts Datorikas fakultātē

Dekānā pilnvarotā persona:

Darnbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

Komisija sekretārs(e):