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

Abstract

Atslēgvārdi

Saturs

APZĪMĒJUMU SARAKSTS

Nukleotīds

YACC

LEX

AVL

TLV

BNF

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 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. [Vai vajag rast nosaukumu izstrādātajai programmatūrai?]

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.

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"[ATSUACE]. 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, kuri dos iespēju analizēt jau eksistējošu pirmkodu. Tika izstrādāti likumi ar kuru palīdzību pirmkods tiek pārveidots par simboliskām virknēm, un vēlāk par datubāzes ierakstiem.

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.

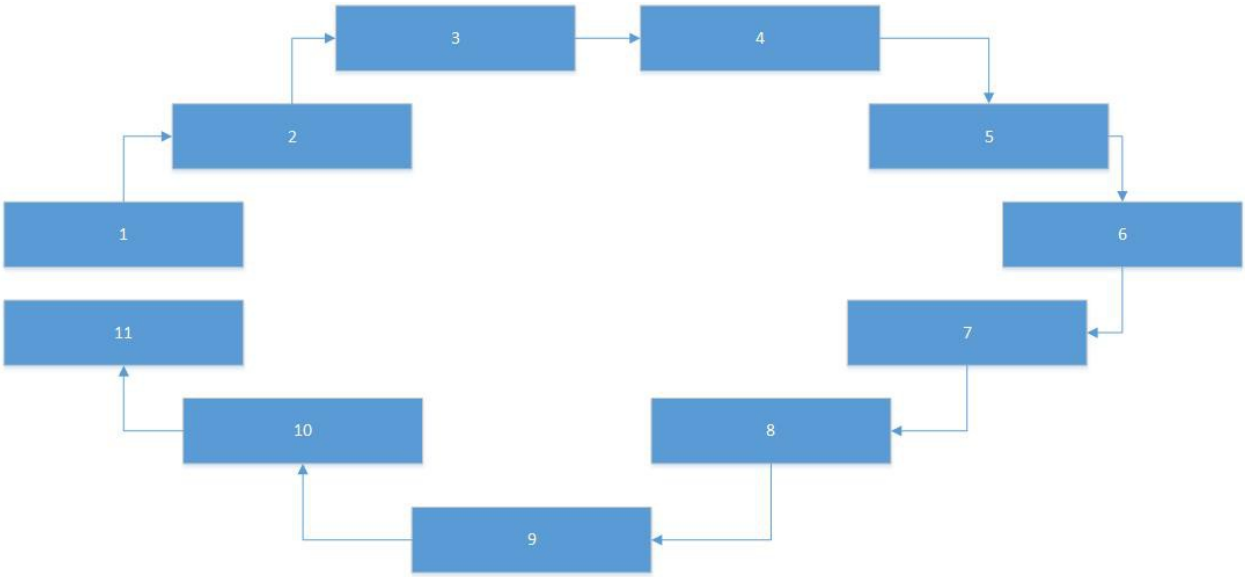
2.2 Moduļu sadalijums pa izpildāmām programmām

Pirmie seši minētie moduļi veido trīs programmatūras failus -

1. lexer.bin – sastāv no LEX YACC ģenerētiem failiem. 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<N>.bin, kur N ir nummurs pēc kārtas. Organisms ir čaulas modulis ap padoto pirmkodu. Tas veic pirmkoda kompilēšanu, startēšanu un pēc pabeigšanas paziņo Pasaulei par darba beigām. Kā var noprast, varbut vairāki šādi moduļi, kuri ir vienlaikus palaisti. Turpmākā izstrādē šie moduļi konkurēs par resursiem.

2.3 Vienkāršots 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.



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. Datu kokā ievietoti dati tiek pārveidoti ar specifisku algoritmu, kura rezultātā padotā koda mainīgo kārtība ir sakārtota noteiktā secībā, kā arī mainīgo vārdi ir standartizēti un tiek izveidotas bloku salīdzināšnas vajdzībām TLV 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, kura ģenerē leksikas analizatorus. Bieži tiek izmantots 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 fails. [ATSAUCE UZ LEX INFORMĀCIJU]

3.2.1 LEX atslēgvārdi

Dotā tabula apraksta uz doto brīdi eksistējošos atslēgvārdus, kuri tiek izmantoti, lai izveidotu leksikas analizatoru. 1. Tabula apraksta iepriekš definētus

3.2.2 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[ATSAUCE] 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 failu daudzumu. Saņemot šo informāciju katrs no saņemtajiem failiem tiks izlaists cauri tālāk aprasktītai procedūrai

Sākumā LEX pārveido visus ieejas datus par atslēgvārdu virkni. Vēlāk tiek izsaukta YACC procedūra, kura izmantojot BNF [ATSAUCE UZ BNF] 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 struktūrā [PIELIKUMS] tiks aizsūtīti pasaulei uz apstrādi. Šis pats process ar apstrādi tiks uzsākts ar nākošo pārbaudes failu. Tāds cikls turpinās līdz visi faili tika pārtulkoti datu struktūras un aizsūtīti pasaulei. Pēc šī brīža lexer.bin paziņo pasaulei, ka tas esot pabeidzis savu darbu un izslēdzas. Ja pasaulei vajdzēs asptrādāt kādu citu failu, tad tā zinās ka vajag atkal sākt visu processu no 0 punkta

3.3 Communicator \ “Communicator”

Komunikācijai paredzēts modulis. Zem šī moduļa ir apkopota visa komunikācija starp dažādiem bināriem failiem. Komunikācija notiek ar SOCKETu palīdzību. Visa saziņa notiek ar iepriekš definētiem ziņojumiem.

3.3.1 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 failu saraksts, kurš tiek nosūtīs no world.bin uz lexer.bin. Uz doto brīdi šis ir vienīgais veids, kā lexer.bin var uzzināt par testējamo failu eksistenci. Trešais ziņojums nāk no lexer.bin puses uz world.bin. Tas satur datus par tikko nolasīto pirmkoda failu. 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.

3.3.2 Ziņojumu struktūra

Savienojuma eksistences un komunikācijas pabeigšanas ziņojumi nesatur nekādus papildus datus. Testējamo failu saraksts un nolasīto datu struktūru saraksts ir sadalīti ierakstos. Katrs ieraksts var satūrēt atslēgu, vārdu un vērtību. Failu saraksta gadijumā atslēgas un vērtības lauki netiek izmantoti. Kad tiek sūtīti dati par pirmkoda failu 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.

3.3.3 Tipiska kumunikācija

Komunikācija notiek starp visiem bināriem failiem. ATTĒLS X eparāda komunikāciju starp world.bin un lexer.bin, nekādiem citiem, papildus, ziņojumiem nevajadzētu eksistēt starp dotiem moduļu kopumiem.

3.4 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.

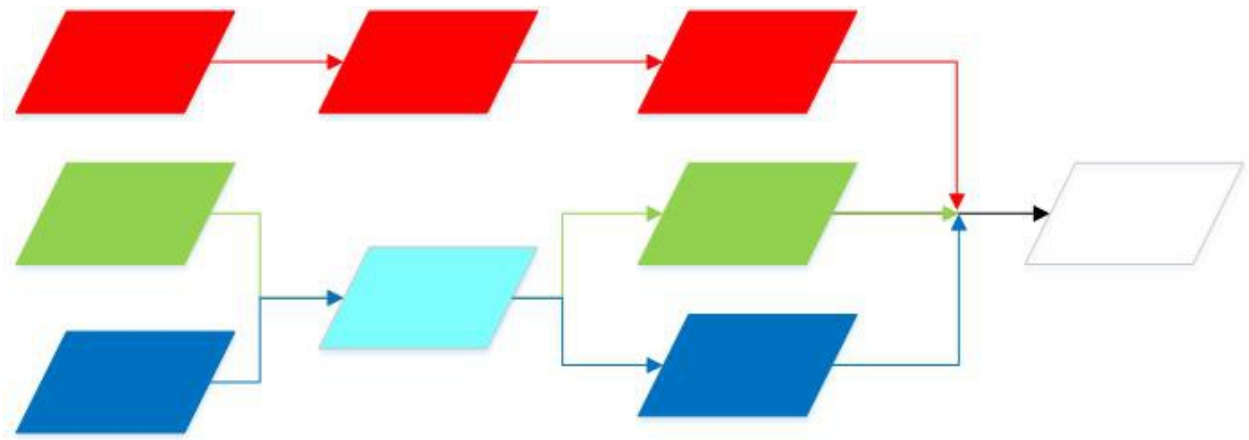
Šis process ir ļoti svarigs, jo visas pārējās darbības notiek jau ar datubāzes datiem un sākotnējais ziņojums par datu struktūra tiks dzēsts. Ja šī soļa laikā tiks ielaista sasaistes kļūda, tas var rast programmai citu priekšstatu par pārbaudāmo pirmkodu.

3.5 Datubāze \ “Primal”

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

3.5.1 Datu glabāšanas veids

Par AVL koku struktūru un algoritmiem var izlasīt rakstā “Algoritms par informācijas organizēšanu”[ATSUACE] 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.



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;*”

3.5.2 Nukleotīdu struktūra

Nukleotīdi sastāv no vairākiem datiem, daļa no informācijas ir obligāta, citi lauki tiks izmantoti tikai noteikta veida nukleotīdiem. Pati struktūra ir atrodama 1. 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 kādam (vai kādiem failiem. Name ir tiešais mainīgā nosaukums. *Type* un *subtype* nosaka nukleotīda tipu, tas 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.

Subtype apraksta konkrēta tipa apkštipu. Tieši šis lauks pasaka tiešo nukleotīda tipu. Šo tipu dažādās iespējas ir aprakstītas 2. Pielikumā. *Pareng* 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.

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 TLV simboliskā virknē.

Lauks *nucleobase_len* apraksta cik daudz nukleotīdu ir aprakstīts nucleobase laukā. Šis lauks izpilda TLV galvenes lomu. 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. Pēc šī simbola seko vētības garuma apzīmējums un, visbeidzot, pati vērtība

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.

3.5.3 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 lauku salīdzināšanu.

3.6 Analizētājs \ “Analyser”

3.7 Organisms \ “Organism”

Dotais modulis tiek izmantots, lai pārbaudītu atsevišķa pirmkoda darbību. Šī moduļa galvenais mērķis ir mēģināt nokompilēt un vēlāk

4 DATU APSTRĀDE\ANALĪZE

Kādā veidā tiek apstrādāti saņemtie dati par struktūrām no LEX\YACC. Datu glabāšanas un salīdzināšanas procesi. Dotās nodaļas mērķis ir rast atbildi kā tiks noteikts ka divi pirmkoda faili ir īstenībā viena un tā paša algoritma realizācija.

Uz doto brīdi ko konkrētu ielikt otrā līmeņa virsrakstos jo dotais solis vēl netika uzprogrammēts. Tas atbilst arī nākošiem pirmā līmeņa virsrakstiem.

5 PROGRAMMAS IZVADA APSTRĀDE

Apraksts par to kāda informācija tiek glabāta, tās izvades veidi. Darbs ar saskarni, kādu nu tā būs tas ir laika jautājums.

6 SALĪDZINĀJUMS AR LĪDZĪGU PORGRAMMATŪRU

Otrā līmeņa virsraksti varētu būt konkrētas programmas ar līdzīgu funkcionalitāti.

7 REZULTĀTI.

Kas ir sanācis un nākotnes plāni attiecībā uz doto darbu.

8 IZMANTOTĀ LITERATŪRA

Informācija par LEX\YACC - LEX & YACC TUTORIAL

Why Source Code Analysis and Manipulation Will Always Be Important

Один алгоритм организации информации // Доклады АН СССР. 1962. Т. 146, № 2. С. 263–266.

Savu kursa darbs(?)..

OTHER

1 PIELIKUMS.

1. Tabula

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

Maska	**Atslēgvārds**
"/*"	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 */

2 PIELIKUMS. NUKLEOTĪDA STRUKTŪRA

```
typedef struct nucleotide_s {
    char file[FILE_NAME_MAX_LEN];
    char name[NUCLEOTIDE_NAME_MAX_LEN];
    nucleotide_type_e type;
    nucleotide_u subtype;
    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 id;

    nucleotide_s *parent
    nucleotide_s *sibling

    size_t len;
    char *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