Latvijas Universitāte
Datorikas fakultāte

Ar regulārām izteiksmēm paplašinātu gramatiku dinamiska parsēšana.

Bakalaura darbs

Autors
Jūlija Pečerska

Vadītājs
Guntis Arnicāns
LU docents

Rīga, 2012

Anotācija

Anotācijas teksts latviešu valoda.

Abstract

The text of the abstract in English.

1. Termini un apzīmējumi

Šeit būs visādi termini, kas ir lietoti darbā un kurus vajag paskaidrot. Šeit arī (ja būs nepieciešamība) būs aprakstīti apzīmējumi, kas lietoti darbā.

2. Ievads

Mūsdienīgo programmēšanas valodu sintaksi nevar aprakstīt ar viennozīmīgām bezkonteksta gramatikām. Tāpēc vairākumam valodu parsētāji ir rakstīti ar rokām, uzmanīgi risinot gramatikas konfliktus. Un kaut arī eksistē parsētāju ģeneratori (piem. ANTLR), kas ar neierobežotu ieskatu kodā var atrisināt gramatikas likumu konfliktus, bieži vien to lietošanas sarežģītība ir salīdzināma ar paša parsētāja rakstīšanu.

Tomēr ievērojamas problēmas parādās tad, kad ir nepieciešams pievienot valodai jaunas konstrukcijas. Tas nozīmē, ka parsētāji ir jāparaksta tā, lai iekļautu jaunos likumus un atrisināt jaunus konfliktus. Ir pašsaprotami, ka gadījumos, kad valoda mainās radikāli, tas būs nepieciešams. Tomēr nelielu izmaiņu gadījumā, it īpaši tādu, kas atvieglo programmētāja darbu, varētu iztikt arī bez tā, atļaujot programmētajam pašam pielāgot valodas sintaksi savam vajadzībām.

Dotais darbs apskata sistēmu, kas ļauj dinamiski paplašināt programmēšanas valodu sintaksi un par pamata principu šīs sistēmas izstrādē ir ņemts pašmodificējamo gramatiku jēdziens. Dinamiski modificējamo gramatiku realizējamība ir aprakstīta dažādos rakstos, tomēr vispārīgā gadījumā šāda tipa gramatikas var nekontrolējami mainīties, izveidojot pavisam citu gramatiku sākotnējās gramatikas vietā. Līdz ar to neierobežotas modifikācijas var izraisīt neprognozējamas sekas. Piedāvātā sistēma ir balstīta uz sakarīgu gramatikas modifikāciju iespēju ierobežojumu, kā arī uz tipu sistēmas, kas ļaus pārliecināties, ka modifikācijas ir korektas. Lai kontrolēt modifikāciju procesu sintaktiskās izmaiņas tiks apskatītas ka dinamisks priekšprocesēšanas variants. Sistēmas galvenā īpašība ir tas, ka pēc gramatikas transformācijas izpildes modificētais izejas kods būs garantēti atpazīstams ar sākotnējo gramatiku.

Aprakstāmās sistēmas ideja ir radusies programmēšanas valodas Eq kompilatora izstrādes laikā bet tā nav piesaistīta pie kādas programmēšanas valodas, bet gan pie konkrēta parsētāju tipa. Perspektīvā tā var tikt lietota jebkurai valodai, kuras parserim piemīt noteiktas īpašības un piedāvāt šai valodai pašmodificēšanas iespējas.

Lietojot nelielu funkcionālu valodu šī sistēma ļaus izveidot jaunas gramatiskas konstrukcijas no jau eksistējošās programmēšanas valodas bāzes funkcionalitātes.

FIXME: *Šeit būs paša prototipa apraksts, kad prototips būs tomēr gatavs.*

Šī dokumenta organizācija ir sekojoša. Nodaļa 2 ievieš un paskaidro galvenos jēdzienus, kas vajadzīgi, lai aprakstīt sistēmu. Nodaļa 3 pamato, kāpēc šī problēma ir aktuāla un piedāvā citus risinājuma piemērus ar pamatojumiem, kāpēc tomēr ir vajadzīga cita pieeja. Nodaļa 4 vispārīgi apraksta izstrādājamo sistēmu un tās galvenās īpašības. Nodara 5, savukārt, apraksta prototipu, rīkus un algoritmus, kas tika lietoti izstrādē. Tā arī pamato, kāpēc daži jau gatavie risinājumi nav lietojami šajā gadījumā. 6. nodaļā ir aprakstītas prototipa iespējas un darba izstrādes rezultēti, bet 7. nodaļa apraksta darba secinājumus. Tālāk darbā ir literatūras saraksts, atzinības un pielikumi (prototipa koda gabali un darba piemēri).

3. Problēmas pamatojums

Bieži valodas tādas ka C un C++ ievieš jaunas sintaktiskas konstrukcijas. Dažreiz tas tiešām ievērojami modificē pašu valodu, bet ļoti bieži tēs ir tikai sintaktiskas. Bet tik un tā tās izraisa nepieciešamību pārrakstīt kompilatoru, jo parasti valodām ir ierobežots sintakses modifikāciju atbalsts, vai arī tas vispār neeksistē.

Problēmas no valodas C

4. Iepriekšējās zināšanas (ievads #2)

Šajā nodaļā ir aprakstīti galvenie jēdzieni, kas nepieciešami darba izpratnei un kas lietoti darba izstrādes gaitā.

4.1. Bezkonteksta gramatikas

Bezkonteksta gramatika satur vārdnīcu no simboliem un pārrakstīšanas likumu kopu. Vārdnīca sastāv no termināliem un neterminālem simboliem, un viens no netermināļem ir gramatikas sākuma simbols. Pārrakstītājas likumi ir izskatā A b, kur A ir viens no netermināliem simboliem, bet b ir neterminālu un terminālu simbolu virkne. Kad kāda likuma kreisē puse parādās apstādāmo simbolu rindā, rinda var tikt pārrakstīta aizvietojot kreisēs puses netermināli ar labo likuma daļu. A b parāda, ka A var tikt pārveidots virknē b atkārtoti pārrakstot to lietojot gramatikas likumus. Visu terminālu simbolu virkņu kopa ir saukta par gramatikas ģenerēto valodu. [1] Programmēšanas valodas ir jēdzienu sistēma, kas ļauj aprakstīt algoritmus. Šai sistēmai jābūt viennozīmīgi aprakstāmai un saprotamai programmētajam. Tātad ir nepieciešams apraksts, kas ļauj saprotami un pārskatāmi izveidot

bāzes struktūras valodai. Bezkonteksta gramatikas izgudroja N. Homskis, kas plānoja lietot tos lai aprakstītu reālās cilvēku valodas. Šinī jomā bezkonteksta gramatikas netiek lietotas, jo dabiskās valodas ir pārāk sarežģītas, tomēr šīs gramatikas tiek lietotas lai aprakstītu programmēšanas valodu sintaksi. Programmēšanas valodas globālā līmenī nav kontekst-neatkarīgas, bet tomēr tēs ir neatkarīgas lokāli, un kaut arī ne visas programmēšanas valodu īpašības var aprakstīt ar bezkonteksta gramatikām, tos ir ērti lietot lai parādīt valodas konstrukciju struktūru. Svarīgāka bezkonteksta gramatiku īpašība ir tas, ka tos var mehāniski pārveidot parsētājos, kas ir sistēma, kas skenējot programmas tekstu izveido programmas struktūru. Šī struktūra tālāk ir reprezentēta koka veidā un var tikt kompilēta izpildāmā kodā. [2]

FIXME: *Vienkrāsas valodiņas gramatikas piemērs (Vai to vispār vajag?)*

(Zemāk - šīs ir ka piemērs ko nevar, es neplānoju skaidrot visu, bet ar šo es gribēju parādīt, ka tiešām ne visu var.) Starp īpašībām, kuras nevar aprakstīt ar bezkonteksta gramatikām ir leksiskais tvērums (lexical scope) un statiskā tipizācija (static typing).

FIXME: *Piemēram, viena no valodas īpašībām, ko nevar aprakstīt ar bezkonteksta gramatikām ir tipu sakritības jēdziens.* Piemēram kodu šādā fragmentā: int a; a = 3.4; ar bezkonteksta gramatikām izsekot nevarēs, jo par to gramatikas līmenī ir zināms tikai tas, ka tas ir kaut kāds identifikators, bet pie kura tipa tas pieder, zināms nav.

4.2. Parsētāji

Vairākums parsētāju mūsdienās aktuālākam valodām (piemēram C/C++) ir rakstīti manuāli. Parsētāju tipi - LR, LL, to trūkumi

4.3. Regulārās izteiksmes

Regulēs izteiksmes, kas tie ir un ko ar tām var darīt.

4.4. Priekšprocesori

Varbūt šī nodaļa nav vajadzīga, atkarīgs no tā, vai būs nodaļa 5.2. Kas tie ir un to iespējas. To sakars ar izstrādājamo sistēmu, kāpēc šī sistēma nav priekšprocesors, bet gan kaut kas cits.

4.5. Tipu teorija (?)

Varbūt šī nodaļa nav vajadzīga? īss tipu teorijas pārskats

5. Dinamiskas gramatikas

Dinamiskas vai adaptīvās gramatikas ir gramatiskais formālisms, kas ļauj modificēt gramatikas likumu kopu ar gramatikas rīkiem. [1]

Dinamiskas gramatikas, kas tēs ir. Fakti par to, ka tēs jau ir pētītas un reāli implementējamas un lietojamas. Reālais labums no tām.

FIXME: *No otras puses kāpēc tēs daudz nepētīja un daudz reāli nelieto.* Tēs vispārīgā gadījumā ir nekontrolējamas.

6. Par izstrādājamo sistēmu (ievads #3)

Ka var redzēt no iepriekšējās nodaļas, pašmodificējošās gramatikas ir diezgan sarežģīts rēks, kas kaut arī ir ļoti lietderīgs, mūsdienās gandrīz netiek lietots. Tas netiek lietots savas sarežģītības dēļ un dēļ tā, ka vispārīgā gadījumā pašmodificējošo gramatiku ir ļoti grūti kontrolēt. Ļaujot neierobežoti modificēt gramatiku mēs varam nonākt pie gadījuma, kad sākotnējā gramatika tiek izmesta ārā, bet tās vietā parādās cita, pilnīgi jauna. Tas netiek lietots savas saredzamības dēj un dēļ tā, ka vispārīgā gadījumā pašmodificējošo gramatiku ir ļoti grūti kontrolēt. Ļaujot neierobežoti modificēt gramatiku mēs varam nonākt pie gadījuma, kad sākotnēja gramatika tiek izmesta ārā, bet tās vietā parādās cita, pilnīgi jauna. Šādā gadījumā šīs jaunās gramatikas adekvātumu un korektību nevar garantēt.

Šīs darbs apraksta iespēju izveidot pašmodificējošo kodu ar funkcionālās makro valodas palīdzību. Šī makro valoda ļaus izveidot jaunas valodas konstrukcijas no jau eksistējošām vienībām. Parsētāja darba laikā makro sastapšanas reizes tiks pārrakstītas uz kodu ar attiecīgu struktūru, kas var tikt atpazīti ar valodas sākotnējo gramatiku. Tātad šī sistēma ļaus modificēt gramatiku nebojājot jau eksistējošo sintaksi. Neko pavisam jauno šī makro sistēma nejaus izveidot, lai paliktu savietojamība ar sākotnējo gramatiku, tomēr tā ļaus atvieglot programmētāja darbu dodot iespēju aizstāt kodā sarežģītas konstrukcijas ar vienkāršākām.

6.1. Origins - Eq

Valoda Eq tiek izstrādāta (kur?). Šīs valodas sintakse bāzējas uz LaTeX teksta procesora sintakses, kas ir standarts priekš zinātniskām publikācijām. Konsekventi programma, kas rakstīta valodā Eq ir korekti interpretējama ar LaTeX procesoru. Tajā pašā laikā Eq programma varēs tikt kompilēta vairākumam mūsdienīgu arhitektūru.

FIXME: *Saite uz Eq projektu*

Šeit būs pavisam nedaudz informācijas par to, kas ir Eq, kāpēc vispār parādies šī ideja.

6.2. Sintakse un darbība

fixmeŠeit būs makro sintakses piemērs

6.3. Pieejas universālums

Kaut arī ideja un pieejas izstrāde sākās ar valodu Eq, tā nav piesaistīta tieši šai valodai. Visspēcīgāka šīs sistēmas īpašība ir tas, ka tā ir universāla un var tikt pielietota jebkādam parserim kas atbilst dažiem nosacījumiem. Par parsētāja modeli

FIXME: *Kāpēc ir izvēlēts tieši LL(k) parsētāji?*

6.4. Sistēmas īpašības

Šī nodaļa aprakstīs, kā mēs gribam realizēt gramatikas pašmodificēšanos, lai izmaiņas būtu kontrolētas. Mēs gribam norobežot modificēšanas iespējas

6.4.1. Pārrakstīšanas sistēma

Match sistēma, atļauto regulāro izteiksmju sintakse. Sintakse ir viegli paplašināma. Konteksti

6.4.2. Tipu sistēma

Kā tiks pārbaudīti tipi.

7. Prototipa realizācija

Pagaidām sistēma nav ieviesta Eq kompilatorā, bet atrodas prototipēšanas stadijā.

FIXME: *Šeit tiks aprakstītas izstrādātā prototipa īpašības un izvēlētās pieejas.*

7.1. Pieejas izvēle

7.2. Lietotie algoritmi

Determinizācija, Minimizācija, Apvienošana.

7.3. Kāpēc tieši šāds risinājums

???

Kāpēc šīm uzdevumam neder jau eksistējošas regulāro izteiksmju bibliotēkas. Kāpēc neder vispārpieņemtie automātu apvienošanas algoritmi. Regulāro izteiksmju dzinēji strādā ar tekstu, nevis ar tokeniem, nav vērts mēģināt pielāgot. Automātu apvienošana - visur aprakstītās pieejas nesaglabā, pie kāda no automātiem pieder katrs stāvoklis, it īpaši akceptējošie stāvokļi. Mums ir svarīgi zināt, kāds no automātiem ir akceptēts, jo no tā ir atkarīgs, kura no produkcijām tiks lietota.

8. Rezultēti

8.1. Prototipa īpašības

Tika izstrādāts prototips, kas parada, ka šāda sistēma var tikt implementēta.

FIXME: *Šeit droši vien jāapraksta vairāk par beigu prototipa versiju, par to, ko viņa varēs darīt.* Cik tā ir efektīva?

8.2. Salīdzinājums ar priekšprocesoriem

FIXME: *Varbūt šī nodaļa nav vajadzīga.* Ko sistēma var un ko nevar salīdzinājumā ar priekšprocesoriem.

9. Secinājumi

Tālāk darbs tik turpināts (šeit var pārfrāzēt Conclusions no raksta melnraksta).

10. Random thoughts

10.1. Saistība ar priekšprocesoriem

Ir divu veidu priekšprocesori - leksiskie un sintaktiskie. Leksiskie priekšprocesori tiek palaisti pirms pirmkoda parsēšanas un nezin neko par apstrādājamas valodas sintaksi (piem. C/C++ priekšprocesors). No otras puses sintaktiskie priekšprocesori tiek palaisti pēc parsera darbības un apstrādā sintaktiskos kokus, ko uzbūvē parsētājs. Dēļ aprakstāmās sistēmas īpašībām šajā darbā netiks apskatīti sintaktiskie priekšprocesori, jo sistēmas īpašība ir tāda, ka līdz tas darba izpildei parsētājs nevar uzbūvēt sintaktisko koku.

Bet leksiskie priekšprocesori pēc savām īpašībām ir tuvi aprakstāmai sistēmai. Ar makro valodu palīdzību tiem tiek uzdoti koda pārrakstīšanas likumi, un kods tiek pārveidots attiecīgi tēs. Bet leksisko priekšprocesoru vislielākais trūkums ir tas, ka tie apstrādā tekstu pa tokeniem neievērojot izteiksmju un konstrukciju struktūru. Piemēram, apskatīsim šādu izteiksmi - `|(a|b)+c|`, kurai vajadzētu tikt pārveidotai uz `abs((a|b)+c)`. Ar tādu makro sistēmu, kas neievēro koda struktūru, tātad neievēro to, ka patiesībā `(a|b)+c` ir atomāra konstrukcija izteiksmē, šādu koda gabalu pareizi pārrakstīt nevarēs. Vidējā `|` zīme sabojās konstrukciju un priekšprocesors nevarēs apstrādāt šādu gadījumu.

Priekšprocesoru var iemācīt apstrādāt šāda veida konstrukcijas un atpazīt tos, ka atomārās izteiksmes. Bet tas nozīmēs, ka priekšprocesoram būs jāzina apstrādājamas valodas sintakse, kas neatbilst priekšprocesora lomai kompilēšanas procesā un nozīmē ka būs divreiz jāimplementē sintakses atpazīšana.

Otrā problēma ar šāda tika priekšprocesoriem ir tas, ka tie strādā ārpus programmas kontekstiem. Tas nozīmē, ka konteksta sākuma tokens (`{` C/C++, Java un citu valodu gadījumā) tiek uzskatīts par parastu tekstu un var tikt pārrakstīts. Loģiskāk būtu, ja kontekstu makro tiktu mantoti līdzīgi ka mainīgie makro, kas ir specifiski kontekstam būtu ar lielāku prioritāti ka tie, kas definēti vispārīgākā kontekstā.

10.2. Our goals

Apskatāmās sistēmas 2 galvenie mērķi ir dot iespēju ieviest jaunas konstrukcijas un tajā pašā laikā saglabāt korektu jau iepriekšeksistējošās sintakses apstrādi.

10.3. Programmas konteksti

Programmas konteksts (pēc Wikipedia) ir vismazākā datu kopa, ko vajag saglabāt programmas darbības pārtraukuma gadījumā, lai varētu atjaunot programmas darbu. Bet pašas programmas iekšienē var eksistēt lokālie konteksti, ko ievieš, piemēram, figūriekavas C/C++ gadījumā. Tad mainīgie, kas tiek definēti vispārīgā programmas kontekstā (globālie mainīgie), var tikt pārdefinēti mazākajā kontekstā (piemēram, kaut kādas funkcijas vai klases robežās) un iegūst lielāku prioritāti. Tas nozīmē, ka ja tiek lietots šāds pārdefinēts mainīgais, tas tiek uzskatīts par lokālu un tiek lietots lokāli līdz specifiska konteksta beigām, nemainot globālā mainīgā vērtību.

Konteksta piemērs:

```
int a = 0;
int b = 1;
int main() {
int a = 2;
a++;
b += a;
}
```

Šajā piemērā `a` ir definēta gan globāli, gan lokāli. Kad tiek izpildīta rindiņa `a++;`, lokāla mainīgā vērtība tiks samazināta uz 3, jo `a` ir pārdefinēts ar vērtību 2. Globālais `a` tā ara paliks ar vērtību 0. Un kad tiks izpildīta rindiņa `b += a;`, `b` pieņems vērtību 4. Konteksta iekša tiks samainīta globālā mainīgā `b` vērtība, jo tas netika pārdefinēts.

Tālāk termins koda konteksts tiks lietots tieši šajā nozīmē.