Latvijas Universitāte
Datorikas fakultāte

# Ar regulārām izteiksmēm paplašinātu gramatiku dinamiska parsēšana.

**Bakalaura darbs**

Autors
*Jūlija Pečerska*

*Vadītājs*
*Guntis Arnicāns*
*LU docents*

Anotācija

Rīga, 2012

## Anotācija

Anotācijas teksts latviešu valoda.

Abstract

**Anotācija**

The text of the abstract in English.

# 1. Termini un apzīmējumi

Šeit būs aprakstīti termini, saīsinājumi un, ja būs nepieciešamība, apzīmējumi.

# 2. Ievads

Mūsdienīgo programmēšanas valodu sintaksi nevar aprakstīt ar vienno-zīmīgām 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 apraksta problēmu un stāsta, kāpēc šī problēma ir aktuāla. Tā arī 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. Iepriekšējās zināšanas (ievads #2)

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

## 3.1. 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` $\Rightarrow$ `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 nentermināli ar labo likuma daļu. `A` $\Rightarrow$ `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. [3] 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. [1]

**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.

### 3.2. Parsētāji

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

**FIXME**: *paskaidrot, no nozīmē reducēt gramatikas likumus*

### 3.3. Regulārās izteiksmes

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

### 3.4. Priekšprocesori

Kas tie ir un to iespējas.

### 3.5. Dinamiskas gramatikas

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

Dinamiskas gramatikas, kas tās ir. Fakti par to, ka tās 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.

### 3.6. Tipu teorija (?)

Varbūt šī nodaļa nav vajadzīga? īss tipu teorijas pārskats

# 4. Problēmas apraksts

Ļoti bieži mūsdienīgas valodas ievieš jaunas sintaktiskās konstrukcijas lai paplašinātu valodas iespējas un lietojamību. Dažreiz šīs izmaiņas izraisa ievērojamas valodas modifikācijas, bet dažreiz šīs izmaiņas ir tikai tā sauktais sintaktiskais cukurs, *syntactic sugar*, konstrukcijas kas tiek pievienotas valodai tikai lai padarītu valodu lasāmāku un patīkamāku cilvēkam. Šīs konstrukcijas nemaina valodas funkcionalitāti, bet gan atvieglo tās lietošanu. Labs sintaktiskā cukura piemērs ir C valodas konstrukcija `a[i]`, kas patiesībā ir `*(a + i)`.

Bet tik un tā jebkāda tipa izmaiņas prasa arī valodas gramatikas izmaiņas, kas vairākumā gadījumu nozīmē parsētāja vai kompilātora pārrakstīšanu. Lai tas nebūtu nepieciešams, valodai jāsatur sintakses modifikācijas atbalsts, kas parasti vai nu ir ļoti ierobežots, vai arī neeksistē vispār.

Šīs darbs pieņem, ka viena no ērtākām pieejām, kā varētu izvairīties no kompilatora pārrakstīšanas nelielu valodas gramatikas izmaiņu gadījumā, ir adaptīvo gramatiku principa pielietošana. Tas nozīmē, ka valodai jāsatur konstrukcijas, kas parsēšanas laikā var modificēt un paplašināt pašas valodas sintaksi.

Viens no metodēm, kā varētu izpildīt pašmodificēšanas uzdevumu ir izveidot **kross-kompilatoru**, kas transformētu jauno sintaksi tā, lai standarta kompilators to varētu atpazīt. Bet šīs metodes problēma ir tas, ka lielākas daļas moderno valodu sintaksi ir neiespējams noparsēt lietojot automātiskos rīkus. Zemāk ir piedāvāti daži piemēri gadījumiem no populāras valodas C, kad automātiskā parsēšana ir neiespējama.

1. Valodā C lietotājs var nodefinēt patvaļīgu tipu lietojot konstrukciju `typedef`. Šāda veida iespēja padara neiespējamu šādas izteiksmes apstrādi `(x) + 5`, ja vien mēs neesam pārliecināti, kas ir `x` - tips vai mainīgais. Ja `x` ir tips, tad šī izteiksme pārveido izteiksmes `+ 5` vērtību uz tipu `x`. Ja `x` ir mainīgais, tad šī izteiksme nozīmē vienkāršu mainīgā `x` un vērtības `5` saskaitīšanu.

2. Pieņemsim, ka ir iespēja paplašināt C valodas sintaksi ar infiksu operatoru `++` un pierakstīt konstanšu masīvus `[1, 2, 3]` veidā. Tad

izteiksme `a ++ [1]` būtu nepārsējama, jo eksistē vismaz divi to interpretācijas veidi. Tas varētu tikt saprasts ka postfiksā operatora `++` pielietošana mainīgam `a` un tad `a` indeksēšana ar `[1]`. Vai arī tas varētu būt divu masīvu `a` un `[1]` konkatenācija.

Dažreiz arī programmatūras koda dalīšana pa tokeniem ir atkarīga no šī koda konteksta, kas padara ne tikai parsēšanas procesu, bet arī leksēšanas procesu neautomatizējamu.

Tas nozīmē, ka **kross-kompilatora** arī būs jāraksta manuāli, risinot eksistējošās gramatikas konfliktus, un oriģinālvalodas ievērojamu izmaiņu gadījumā būs jāpastrādā abi kompilatori, kas nozīmē divreiz vairāk darba.

# 5. Par izstrādājamo sistēmu (ievads #3)

Ka var redzēt no iepriekšējās nodaļas, pašmodificējošās gramatikas ir diezgan sarežģīts rīks, 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.

## 5.1. 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.

## 5.2. Sintakse un darbība

fixmeŠeit būs makro sintakses piemērs

## 5.3. 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?*

## 5.4. Sistēmas īpašības

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

### 5.4.1. Pārrakstīšanas sistēma

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

### 5.4.2. Tipu sistēma

Kā tiks pārbaudīti tipi.

## 5.5. Saikne 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.

# 6. Prototipa realizācija

Lai ilustrētu šādas sistēmas izstrādes iespējamību, tika izstrādāts prototips, kas parada, ka šāda sistēma var tikt implementēta. Šī nodaļa apraksta prototipa vispārīgās īpašības un pieejas, kas tika lietotas tā realizācijā. Prototips vienkāršības un izstrādes ātruma dēļ tika rakstīts Python valodā, kas ir skriptu valoda, un tāpēc prototips ir viegli palaižams un atkļūdojams uz jebkuras mašīnas ar uzstādītu 2.7.0 Python versiju.

## 6.1. Vispārīgā pieeja

Šī apakšnodaļa apraksta sistēmas prototipa darbību virkni un prototipa iespējas uz doto brīdi. Šeit ir tikai vispārīgi aprakstīta darba gaita, bez pamatojumiem, kāpēc šāda rīcība ir izvēlēta, bet apakšnodaļā 6.4. tiks sīkāk aprakstīts, kāpēc tas tika izveidots tieši šādi un ne citādi.

Prototips imitē darbu reālajā vidē, no mockup leksera klases lasot pa vienam tokenus no ieejas plūsmas. Ir viens vispārējā rakstura tokens, no kura tiek mantotas dažādas tokenu apakšklases. Piemēram, eksistē tokena klase izteiksmei - `t_expr`, kuras apakšklase ir reāls skaitlis `t_real`. Tajā pašā laikā `t_real` ir virsklase veselu skaitļu klasei `t_int`. Tokeniem var būt saturs, piemēram regulārā izteksme `t_id('f')` sagaidīs tieši identifikatoru `f`, bet izteksme `t_id()` sagaidīs jebkuru identifikatoru. Šie tokeni nav reāli, to klases tika izvēlētas uzskatamības pēc, bet patiesībā tokenu skaitam un hierarhijai nav nozīmes. Vienīgais, ko vajag zināt sistēmai, lai veiksmīgi strādātu, ir to mantošanas struktūra un tokenu vērtību salīdzināšana, lai tā zinātu sakritības attiecības starp regulārās izteiksmes tokeniem un ieejas plūsmas tokeniem.

Tā kā šī sistēma nav parsētājs, tā neapstrādā tokenus, kas neattiecās uz sistēmas darbu. Tas nozīmē, ka kamēr sistēmā neeksistē neviena regulārā izteiksme, tā palaiž garām tokenus un neapstrādā tos. Tiklīdz tiek sastapts makro sākuma tokens, prototips uzsāk regulārās izteiksmes parsēšanu. Parsēšanas procesā tiek izveidots nedeterminēts galīgs automāts. Tālāk šīs automāts tiek determinēts un minimizēts, tātad katra regulāra izteiksme tiek pārveidota minimālajā determinētā automātā, tātad ir optimizēta pēc

izpildes laika.

Tikko parādās vismaz 2 regulārās izteiksmes, sistēma sapludina kopā to determinētos automātus, kas tiek darīts lai samazinātu tokenu virknes sakrišanas atrašanas laiku. Tā izteiksme, kas tika ielasīta agrāk būs ar lielāku prioritāti nekā tā, kas ir ielasīta vēlāk. Tātad ja secīgi tiks ielasītas 2 izteiksmes `t_id() ( )` un `t_id() ( t_real() * )`, tad ielasot virkni `t_id('f') ( )` tiks akceptēta pirmā izteiksme. Gadījumā, ja izteiksmes tiks ielasītas pretējā secībā, pirmā izteiksme nekad netiks atpazīta, jo otrā izteiksme pārklāj visas pirmās izteiksmes korektās ieejas.

Tālāk, kad atnāk kaut kādi tokeni, kas neatzīmē makro sākšanās, sistēma izpilda pārejas starp sapludinātā automāta stāvokļiem un atceras tokenus, kas jau ir nolasījusi. Tikko kāds no automāta stāvokļiem ir akceptējošs, sistēma aizstāv ielasīto virkni ar citu, kas ir konstruēta pēc akceptētās regulārās izteiksmes noteikumiem. Tad darbs tiek uzsākts no aizvietotās virknes sākuma.

Sistēma turpina darbu tādā pašā gaitā līdz ieejas tokenu virknes beigām.

## 6.2. Izņēmumi

Prototipā pagaidām nav implementēta apstrādes dalīšana pa gramatikas produkcijām, visas regulārās izteiksmes ir sapludinātas vienā automāta. Regulāro izteiksmju dalīšana pa tipiem tiks izstrādāta vēlāk, kad tiks uzsākta integrācija un sadarbība ar reālu kompilatoru. Tā varētu tikt implementēta līdzīgi ka konteksti -

## 6.3. Lietotie algoritmi

### 6.3.1. Determinizācija

**FIXME**: *Write!*

### 6.3.2. Minimizācija

**FIXME**: *Write!*

### 6.3.3. Apvienošana

**FIXME**: *Write!*

## 6.4. Kāpēc tieši šāds risinājums

**FIXME**: *Write!*

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.

# 7. Rezultāti

## 7.1. Prototipa īpašības

Prototips pagaidām netiek integrēts Eq valodas kompilātorā, bet tas tiek plānots tuvākajā nākotnē.

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

## 7.2. Salīdzinājums ar priekšprocesoriem

**FIXME**: *Varbūt šo nodaļu vajag pārsaukt, varbūt tā nav vajadzīga*

Ko sistēma var un ko nevar salīdzinājumā ar priekšprocesoru funkcionalitāti. Vai

# 8. Secinājumi

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

# 9. Random thoughts

## 9.1. Our goals

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

## 9.2. 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.

### 9.3. Parsētāji

Lai parsētājs varētu kļūt par bāzi izstrādājamai transformāciju sistēmai, tam jābūt izstrādātam ar rekursīvas nokāpšanas algoritmiem LL(k) vai LL(*). LL ir viena no saprotamākām parsētāju rakstīšanas pieejam, kas ar lejupejošo procesu apstrādā programmatūras tekstu. Gadījumā, ka gramatika ir labi rakstīta (k, simbolu skaits ieskatam uz priekšu, ir mazs) parsētāja darba ātrum atkarībā no tokenu daudzuma var tuvoties lineāram. [2] Tā kā transformāciju sistēma tiek veidota ka paplašinājums parsētājam, tā prasa lai parsētājs uzvedās zināmā veidā. Zemāk tiks aprakstītas īpašības, kurām jāatbilst parsētājam, lai uz tā veiksmīgi varētu uzbūvēt aprakstāmo sistēmu.

**Tokenu virkne** Parsētājam jāprot aplūkot tokenu virkni ka abpusēji saistītu sarakstu, lai eksistētu iespēja to apstaigāt abos virzienos. Tam arī jādod iespēju aizvietot kaut kādu tokenu virkni ar jaunu un ļaut uzsākt apstrādi no jaunās virknes sākuma.

**Pseido-tokeni** Parsētāji parasti pielieto (reducē) gramatikas likumus ielasot tokenus no ieejas virknes. Pseido-tokens, savukārt, konceptuāli ir atomārs ieejas plūsmas elements, bet īstenībā attēlo jau reducētu kaut kādu valodas gramatikas likumu. Viens no pseido-tokeniem, piemēram, ir tokens izteiksme - `expr`, kas var sastāvēt no daudziem dažādiem tokeniem (piem. `(a+b*c)+d`). Tas nav viens tokens, bet tā ir tokenu grupa, ko atpazīst parsētājs un kas var tikt attēlots ka atomāra vienība.

**Vadīšanas funkcijas** Pirmkārt, mēs prasam, lai katra gramatikas produkcija tiktu reprezentēta ar vadīšanas funkciju (*handle-function*). Ir svarīgi atzīmēt, ka šim funkcijām būs blakus efekti, tāpēc to izsaukšanas kārtība ir svarīga. Šo funkciju signatūrai jāizskatās šādi: `Parser` $\rightarrow$ `(AST|Error)`, tas ir, funkcija ieejā iegūst parsētāja objektu un izejā atgriež abstraktā sintakses koka (Abstract Syntax Tree) mezglu vai arī kļūdu. Šīs funkcijas atkārto gramatikas struktūru, tas ir ja gramatikas produkcija A ir atkarīga no produkcijas B, A-vadīšanas funkcija izsauks B-vadīšanas funkciju.

Katra no šādām funkcijām pēc nepieciešamības implementē arī kļūdu

apstrādi un risina konfliktus starp produkcijām ar valodas apraksta palīdzību.

**Piederības funkcijas** Katrai vadīšanas funkcijai pārī ir piekārtota funkcija-predikāts. Šīs predikāts pārbauda, vai tā vietā tokenu virknē, uz kuru dotajā brīdī norāda parsētājs, atbilst parsētam gramatikas likumam. Šādas piederības funkcijas (*is-function*) izpilde nemaina parsētāja stāvokli.

**Sakrišanas funkcijas** Katras vadīšanas funkcijas darbības sākumā tiek izsaukta tā sauktā sakrišanas funkcija (*match-function*). Sakrišanas funkcija ir transformācijas sistēmas saskarne ar signatūru `(Parser, Production)` $\rightarrow$ `Parser`. Tā pārbauda, vai tā vieta tokenu virknē, uz kuru rāda parsētājs, ir derīga kaut kādai transformācijai dotās produkcijas ietvaros. Ja pārbaude ir veiksmīga, funkcija izpilda sakrītošās virknes substitūciju ar jaunu virkni un parsētāja stāvoklī uzliek norādi uz aizvietotās virknes sākumu. Gadījumā, ja pārbaude nav veiksmīga, funkcija nemaina parsētāja stāvokli, un parsētājs var turpināt darbu nemodificētas gramatikas ietvaros.

Ja izstrādājamās valodas parsētāja modelis atbilst aprakstītām īpašībām, tad uz tās var veiksmīgi uzbūvēt aprakstāmo transformāciju sistēmu un ļaut programmētājam ieviest modifikācijas oriģinālās valodas sintaksē.

# Literatūra

[1] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Forbes D. Lewis. Recursive descent parsers. http://www.cs.engr.uky.edu/∼lewis/essays/compilers/rec-des.html.

[3] John N. Shutt. Adaptive grammars. http://web.cs.wpi.edu/ jshutt/, November 2005.