

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 valodā

Atslēgvārdi: Dinamiskās gramatikas, priekšprocesēšana, makro, regulārās izteiksmes, galīgi determinēti automāti, Python

Abstract

Abstract text in English

Keywords: Dynamic grammars, preprocessing, macros, regular expressions, determinate finite automata, Python

Saturs

1.	Termini un apzīmējumi	3
2.	Ievads	4
3.	Iepriekšējās zināšanas (ievads #2)	6
3.1.	Bezkonteksta gramatikas	6
3.2.	Parsētāji	6
3.3.	Regulārās izteiksmes	7
3.4.	Priekšprocesori	7
3.5.	Tokeni	7
3.6.	Pseido-tokeni	8
3.7.	Programmas konteksti	8
3.8.	Dinamiskas gramatikas	8
3.9.	Tipu teorija (?)	9
4.	Problēmas pamatojums	10
4.1.	Problēmas apraksts	10
5.	Transformāciju sistēma	11
5.1.	Idejas rašanās - valoda Eq	11
5.2.	Parsētāji	12
5.3.	Makro sistēmas sintakse	13
5.4.	Sistēmas īpašības	14
5.4.1.	Funkcionālā valoda T	14
5.4.2.	Tipu sistēma	15
5.5.	Sistēmas sakars ar priekšprocesoriem	15
6.	Prototipa realizācija	17
6.1.	Vispārīgā pieeja	17
6.2.	Lietotie algoritmi	18
6.2.1.	Determinizācija	18
6.2.2.	Minimizācija	18
6.2.3.	Apvienošana	18
6.3.	Realizācijas pamatojums	18
6.4.	Problēmas un izņēmumi	19
6.4.1.	Problēmas	19
6.4.2.	Izņēmumi	19

7.	Rezultāti	20
7.1.	Prototipa īpašības	20
8.	Secinājumi	20
9.	Random thoughts	21
9.1.	Our goals	21
9.2.	Why adaptable grammars are cool	21
9.3.	Why adaptable grammars suck	21

1. Termini un apzīmējumi

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

2. Ievads

Mūsdienīgo programmēšanas valodu sintaksi nevar aprakstīt ar viennozīmīgām kontekstneatkarīgām gramatikām. Tāpēc vairākiem 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 jaunus 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 neprognozējamās 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 kā dinamisks priekšprocesēšanas variants. Sistēmas galvenā īpašība ir tas, ka pēc gramatikas transformācijas izpildes modificētais izejas kods būs atpazīstams ar sākotnējo gramatiku.

Aprakstāmās sistēmas ideja ir radusies programmēšanas valodas Eq kompilatora izstrādes laikā. Tomēr tā nav piesaistīta pie kādas programmēšanas valodas, bet gan pie konkrēta parsētāju tipa. Perspektīvā tā var tikt lietota jebkurai valodai, kuras parsētājam piemīt noteiktas īpašības un piedāvāt šai valodai pašmodificēšanas iespējas. Lietojot makro šablonus sakrišanas atrašanai kodā un vienkāršu funkcionālu valodu atrasto virkņu modifikācijai, šī sistēma ļaus izveidot jaunas gramatiskas konstrukcijas no jau eksistējošās programmēšanas valodas bāzes funkcionalitātes.

Šī darba mērķis ir izstrādāt prototipu, kas pierādīs iespēju izveidot strādājošu šablonu sakrišanas sistēmu, kas strādātu uz leksiskā analizatora izveidotiem tokeniem. Prototipa izstrādes galvenais uzdevums ir atrast efektīvu veidu, kā apstrādāt makro prioritātes, ieejas un izejas no kontekstiem un sakrišanas konstatēšanu.

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 aprakstītu 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. Nodaļa 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, darba izstrādes rezultāti un parādītas testēšanas stratēģijas, bet 7. nodaļa apraksta darba secinājumus.

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 netermināliem simboliem, un viens no netermināliem 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 netermināli 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. [5] 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 tās 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 abstraktās sintakses koka (AST, Abstract Syntax Tree) veidā un var tikt kompilēta izpildāmā kodā. [3]

FIXME: *Vienkāršas 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 pieder, zināms nav.*

3.2. Parsētāji

Vairākums parsētāju mūsdienās aktuālākām 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. Tokeni

Pirmā programmas kompilēšanas fāze ir leksiskā analīze jeb skanēšana. Tās laikā leksiskais analizators lasa ieejas simbolu virkni (programmas izejas tekstu) un veido jēdzīgas simbolu grupas, kas ir sauktas par leksēmām. Katrai leksēmai leksiskais analizators izveido speciālu objektu, kas tiek saukts par tokenu. Katram tokenam ir glabāts tokena tips, ko lieto parsētājs lai izveidotu programmas struktūru. Ja ir nepieciešams, tiek glabāta arī tokena vērtība, parasti tā ir norāde uz elementu simbolu tabulā, kurā glabājas informācija par tokenu - tips, nosaukums. Simbolu tabula ir nepieciešama tālākā kompilatora darbā lai paveiktu semantisko analīzi un koda ģenerāciju. Šajā darbā vienkāršības dēļ tiks uzskatīts, ka tokena vērtības ailītē glabāsies leksēma, ko nolasīja analizators. Tālāk tokeni tiks apzīmēti šādā veidā:

{token-type : token-value}

Nolasīto tokenu virkne tiek padota parsētājam tālākai apstrādei.

Piemēram apskatīsim nelielu programmas izejas koda gabalu - `sum = item + 5`. Šīs izejas kods var tikt sadalīts sekojošos tokenos:

1. `sum` ir leksēma, kas tiks pārtulkota tokenā {id:sum}. `id` ir tokena klase, kas parāda, ka nolasītais tokens ir kaut kāds identifikātors. Tokena vērtībā nonāk identifikatora nosaukums `sum`.
2. Piešķiršanas operators `=` tiks pārveidots tokenā {=}. Šim tokenam nav nepieciešams glabāt vērtību, tāpēc otrā tokena apraksta komponente ir izlaista. Lai atvieglotu tokenu virkņu uztveri šī darba ietvaros operatoru tokenu tipi tiks apzīmēti ar operatoru simboliem, kaut arī pareizāk būtu izveidot korektus tokena tipu nosaukumus, piemēram {assign}.
3. Leksēma `item` analogiski `sum` tiks pārtulkota tokenā {id:item}.
4. Summas operators `+` tiks pārtulkots tokenā {+}.
5. Leksēma `5` tiks pārtulkota tokenā {int:5}.

Tātad izejas kods `sum = item1 + 5` pēc leksiskās analīzes tiks pārveidots tokenu plūsmā {id:sum}, {=}, {id:item1}, {+}, {int:5}. [1]

3.6. Pseido-tokeni

Šī darba ietvaros tiek lietots jēdziens pseido-tokens

Tālāk darbā tiks lietotas šādas notācijās pseido-tokenu aprakstam: `{expr}`, `{id}`, `{int}`, `{real}`. Pseido-tokenu vērtība tiek apzīmēta sekojoši: `{id:foo}`. Šāds apzīmējums nozīmē, ka tas ir tokens ar tipu `id` un ar vērtību `foo`.

3.7. 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 ar 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.

3.8. Dinamiskas gramatikas

Dinamiskas vai adaptīvās gramatikas ir gramatiskais formālisms, kas ļauj modificēt gramatikas likumu kopu ar gramatikas rīkiem. [5]

Dinamiskas gramatikas, kas tās ir. Fakti par to, ka tās jau ir pētītas un reāli implementējamās un lietojamās. Reālais labums no tām.

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

3.9. Tipu teorija (?)

Varbūt šī nodaļa nav vajadzīga? Īss tipu teorijas pārskats

4. Problēmas pamatojums

4.1. 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 kompilatora pārrakstīšanu. Lai tas nebūtu nepieciešams, valodai jāsaturs sintakses modifikācijas atbalsts, kas parasti vai nu ir ļoti ierobežots, vai arī neeksistē vispār.

Šis 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 jāsaturs 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 infīksu 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 postfīksā operatora `++` pielietošana mainīgā `a` un tad `a` indeksēšana ar `[1]`. Vai arī tas varētu būt divu masīvu `a` un `[1]` konkatēnācija.

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.

FIXME: *Continue*

5. Transformāciju sistēma

Ka var redzēt no nodaļas 3.8., 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. Neierobežotas modifikācijas iespējas var arī ieviest tādas gramatikas īpašības, kas neļaus parsētājam pareizi darboties (piemēram kreisā rekursija LL parsētāju gadījumā). Tātad vispārīgā gadījumā jaunās gramatikas pareizību nevar garantēt.

Šis darbs piedāvā uzbūves principus sistēmai, kura dod iespēju programmētājam dinamiski paplašināt valodas iespējas ar makro valodas palīdzību. Šī makro valoda ļaus izveidot jaunas valodas konstrukcijas no jau eksistējošām vienībām ar regulāro izteiksmju un nelielas funkcionālās valodas palīdzību. 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. Nekādas pavisam jaunas konstrukcijas šī makro sistēma neļaus izveidot, lai paliktu savietojamība ar sākotnējo gramatiku, tomēr tā ļaus atvieglot programmētāja darbu dodot iespēju aizstāt kodā sarežģītas konstrukcijas ar vienkāršākām.

Šīs sistēmas galvenais mērķis ir piedāvāt iespēju modificēt valodas sintaksi programmas rakstīšanas gaitā, nebojājot jau eksistējošo konstrukciju darbu. Sistēma ieviesīs pašmodificēšanos uz pārrakstīšanas bāzes, kas vienlaikus nodrošinās modifikācijas un parsētāja nemainīgumu. Tajā pašā laikā sistēma būs stabila pret kļūdām dēļ tā, ka tā strādās tikai konkrētās produkcijas ietvaros un tā, ka tā pārbaudīs tipus jaunizveidotām virknēm.

Tālāk aprakstāmā sistēma tiks saukta par transformāciju sistēmu. Šī nodaļa dos vispārīgu ieskatu transformācijas sistēmas uzbūvē, darba gaitā, aprakstīs transformācijas sistēmas likumu sintaksi un parādīs iespēju pierādīt transformācijas pareizību.

5.1. Idejas rašanās - valoda Eq

Šīs makro transformācijas sistēmas ideja ir radusies valodas Eq (atrodams tiešsaistē - <https://github.com/zayac/eq>) izstrādes gaitā, kurā piedalās cilvēku grupa no Heriot-Watt University (Riccarton, Edinburgh) un Moscow Institute of Physics and Technology (Dolgoprudny, Russia). Šīs valodas sintakse bāzējas uz L^AT_EX teksta procesora sintakses, kas ir standarts priekš zinātniskām publikācijām. Konsekventi programma, kas rakstūta valodā Eq ir korekti interpretējama ar L^AT_EX procesoru. Tajā pašā laikā Eq programma varēs tikt kompilēta vairākus mūsdienīgu arhitektūru.

Tā kā L^AT_EX sintakse ir viegi paplašināma, arī Eq valodas sintaksei izlēma piedāvāt paplašināšanas iespējas. Makro sintakse bez kaut kādas šablonu sistēmas ir bezjēdzīga, jo ir ļoti ierobežota. Tāpēc tika izvēlēta regulāro izteiksmju šablonu sistēma, kas dod brīvību sakrišanas meklēšanas mehānismam. Tālāk lai apstrādāt regulārās izteiksmes apstrādātos tokenus tika

nolemts izveidot vienkāršu funkcionālu valodu, kas ļaus pārstrādāt tokenu virknes atkarībā no programmētāja izveidotiem šabloniem.

Bet kaut arī ideja un pieejas izstrāde sākas ar valodu Eq, tā nav piesaistīta tieši šai valodai. Visspēcīgāka šīs sistēmas īpašība ir tas, ka tā ir universāla un var tikt pielietota jebkādam parsētājam kas atbilst dažiem nosacījumiem. Par parsētājiem nepieciešamām īpašībām tiks runāts apakšnodaļā 5.2..

5.2. 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 intuitīvi saprotamākām parsētāju rakstīšanas pieejām, kas ar lejupejošo procesu apstrādā programmatūras tekstu. LL parsētājiem nav nepieciešams atsevišķs darbs parsēšanas tabulas izveidošanā, tātad parsēšanas process ir vairāk saprotams cilvēkam un vienkāršāk realizējams, samazinot kļūdu varbūtību. Gadījumā, ka gramatika ir labi rakstīta (k, simbolu skaits ieskatam uz priekšu, ir mazs) LL parsētāja darba ātruma atkarībā no tokenu daudzuma var tuvuoties lineārai. [4]

Tā kā transformāciju sistēma tiek veidota kā 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 kā 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 kā atomāra vienība.

Vadīšanas funkcijas Pirmkārt, mēs prasam, lai katra gramatikas produkcija tiktu reprezentēta ar vadīšanas funkciju (*handle-function*). Ir svarīgi atzīmēt, ka šīm funkcijām būs blakus efekti, tāpēc to izsaukšanas kārtība ir svarīga. Šo funkciju signatūrai jāizskatās šādi: $\text{Parser} \rightarrow (\text{AST}|\text{Error})$, tas ir, funkcija ieejā iegūst parsētāja objektu un izejā atgriež abstraktā sintakses koka (Abstract Syntax Tree) mezglu vai arī kļūdu. Šīs funkcijas atkārtoto 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 pāri ir piekārtota funkcija-predikāts. Šīs predikāts pārbauda, vai tā vietā tokenu virknē, uz kuru dotajā brīdī norāda parsētājs, atbilst

parsētām 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.

5.3. Makro sistēmas sintakse

Makro izteiksmes strādā stingri kaut kādas produkcijas ietvaros, tāpēc makro sintaksē tiek lietoti tipi, kas tiek apzīmēti ar produkciju nosaukumiem. Tipi tiks lietoti lai nodrošinātu tokenu virknes korektību sākotnējās gramatikas ietvaros pēc sintakses izmaiņu ieviešanas. Transformāciju sistēma sastāv no *match* makro likumiem un transformāciju funkcijām. Makro kreisā puse satur regulāro izteiksmi no tokeniem un pseido-tokeniem, kas tālāk tiek izmantota lai atrast tokenu virkni, kurai šī transformācija ir pielietojama. Makro labā pusē ir atrodamas funkcijas, kas ir rakstītas vienkāršā funkcionālā valodā T . Valoda T tiek lietota lai izpildītu transformācijas ar tokenu virknēm, kas tiek akceptēti ar makro funkcijas kreiso pusi.

Apakšnodaļā 5.4. tiks vispārīgi aprakstīta tipu sistēma un funkcionālā valoda T , bet tā kā tās neietilpst šī darba ietvaros, sīkāk tās aprakstītas nebūs.

Vispirms apskatīsim *match* likumus, kas modificē apstrādājamās gramatikas produkcijas uzvedību. *Match* makro sintakses vispārīgu piemēru var redzēt figūrā 1..

$\text{match } [\backslash\text{prod1}] \ v = \text{regexp} \rightarrow [\backslash\text{prod2}] \ f(v)$

1. att. *Match* makro sintakses vispārīgs piemērs

Šis piemērs ir uztverams sekojoši. Ja produkcijas *prod1* sākumā ir atrodama pseido-tokenu virkne, kas atbilst regulārai izteiksmei *regexp*, tad tai tiek piekārtots mainīgais ar vārdu *v*. Mainīgais *v* var tikt lietots makro labajā pusē kaut kādas funkcijas izpildē. Tātad ja tāda virkne *v* eksistē, tā tika aizstāta ar pseido-tokenu virkni, ko atgriezīs $f(v)$ un tālāk reducēta pēc gramatikas produkcijas *prod2* likumiem.

Regulārā izteiksme *regexp* ir vienkārša standarta regulārā izteiksme, kas gramatika ir definēta figūrā 2..


```

regexp      → concat-regex | regexp
concat-regex → asterisk-regex concat-regex
asterisk-regex → unary-regex * | unary-regex
unary-regex  → pseudo-token | ( regexp )

```

2. att. Regulāro izteiksmju gramatika uz pseido-tokeniem

Pagaidām sistēmas prototipa izstrādē tiek lietota šāda minimāla sintakse, bet tālākā darba gaitā tā viegli var tikt paplašināta.

Tagad mēs varam izveidot definētās makro sintakses korektu piemēru. Pieņemsim, ka ērtības dēļ programmētājs grib ieviest sekojošu notāciju absolūtās vērtības izrēķināšanai - $| \{ \text{expr} \} |$. Sākotnējā valodas gramatikā eksistē absolūtās vērtības funkcija izskatā $\text{abs}(\{ \text{expr} \})$. Tad makro, kas parādīts figūrā 3. izdarītu šo substitūciju, ļaujot programmētājam lietot ērtāku funkcijas pierakstu.

```

match [{expr}] v = {|} {expr} {|}
  → [{expr}] {id:abs} {(} {expr} {)}

```

3. att. Makro piemērs #1

Vēl viens korektā makro piemērs: pieņemsim, ka funkcija `replace` ir definēta valodā T ar trim argumentiem, un darba gaitā tā jebkurā pseido-tokenu virknē aizvieto elementus, kas sakrīt ar otro argumentu, ar trešo funkcijas argumentu. Pieņemsim arī, ka mums ir nepieciešams izsaukt funkciju bar ar vienu argumentu, kas ir summa no funkcijas `foo` argumentiem. Šādā gadījumā makro, kas parādīts figūrā 4., izpildīs nepieciešamu darbību.

```

match [{expr}] v = {id:foo} {(} {expr} ( {,} {expr} ) * {)}
  → [{expr}] {id:bar} (replace v {,} {+})

```

4. att. Makro piemērs #2

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. Funkcionālā valoda T

Lai varētu izpildīt atrastās tokenu virknes apstrādi un modificēšanu ir nepieciešams kaut kāds papildus rīks. Par šo rīk utika izvēlēta vienkārša funkcionāla valoda. Tā tika izvēlēta tāpēc, ka tās funkcijām nepiemīt blakusefekti, tātad to izpilde nevarēs samainīt eksistējošos datus. Valoda, kuras funkcijām ir blakusefekti, varētu sabojāt parsētāja darbu.

Šī minimālistiska funkcionālā valoda tika nosaukta par T , un tā tiks lietota lai aprakstītu plānotas tokenu virknes transformācijas iespējas. Valodas gramatika ir parādīta figūrā 5.

```

program      → ( function ) *
function     → id '::' fun_type id id * '=' expr
fun_type     → (type | '(' fun_type ')') '->' fun_type
expr         → id | expr expr | let_expr | if_expr | builtin
let_expr     → 'let' id '=' expr (',' id '=' expr) * 'in' expr
if_expr      → 'if' cond_expr 'then' expr 'else' expr
cond_expr    → 'type' expr '==' type | expr
builtin      → 'cons' expr (expr | 'nil')
              | 'head' expr | 'tail' expr | 'value' expr
              | pseudo_token '[' expr ']' | number
              | + | - | ...
type         → pseudotoken_regexp | int | regexp_t

```

5. att. *T* valodas gramatika

FIXME: *Was ist regexp_t?*

Galvenais šīs valodas pielietojums ir dot iespēju apstaigāt pseido-tokenu virkni, kura tika atzīta par sakrītošu ar atbilstošu šablonu. Lai to darīt, tā dos iespēju lietot rekursiju un dažas iebūvētās funkcijas - saraksta pirmā elementa funkciju `head`, saraksta astes funkciju `tail` un objektu pāra izveidošanas funkciju `cons`. Funkcija `cons` funkcionālo valodu kontekstā strādā kā saraksta izveidošanas funkcija, jo saraksts `list(1, 2, 3)` tiek reprezentēta kā `cons(1, cons(2, cons(3, nil)))`, kur `nil` ir speciāls tukšs objekts.

FIXME: *Paskaidrot sintaksi (if_expr, let_expr, ...)*

Lai būtu iespēja apstādināt rekursiju, šī valoda arī ļaus izpildīt aritmētiskās operācijas ar veseliem skaitļiem. Tas dos iespēju izveidot skaitītājus un izveidot rekursijas izejas nosacījumus.

Ir plānots, ka šī valoda arī ļaus izpildīt daļēju novērtējumu izteiksmēm, tur kur būs nepieciešams. Tas nozīmē, ka valodai jāsaturs saskarne, kas ļaus piekļūt pie tokena vērtības. Šim mērķim ir domāta funkcija `value`, kas ir pielietojama pseido-tokeniem ar skaitlisku vērtību, piemēram, lai dabūt skaitli 5 no pseido-tokena `{int:5}`. Lai izveidotu jaunu tokenu ar izrēķinātu vērtību, tiks lietota sintakse `pseudo_token[value]`, kur `pseudo_token` ir izveidojamā tokena tips un `value` ir vērtība, piemēram, `int[10]`.

Funkcija `type`, savukārt, ļaus pārbaudīt tokenu tipu, kas var būt nepieciešams transformācijas procesā, piemēram, lai atpazīt kādu operatoru.

5.4.2. Tipu sistēma

FIXME: *Write this*

5.5. Sistēmas sakars ar priekšprocesoriem

Ir divu veidu priekšprocesori - leksiskie un sintaktiskie. Leksiskie priekšprocesori tiek palaisti pirms pirmkoda parsēšanas un nezina neko par apstrādājamās 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 $\text{abs}((a|b)+c)$. Ar tādu makro sistēmu, kas neievēro koda struktūru, tāad neievēro to, ka patiesībā $(a|b)+c$ ir atomāra konstrukcija izteiksmē, šādu koda gabalu pareizi apstrādāt 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 tvērumiem. Tas nozīmē, ka tvēruma sākuma tokens ($\{ C/C++, \text{Java un citu valodu gadījumā} \}$) tiek uzskatīts par parastu tekstu un var tikt pārrakstīts. Loģiskāk būtu, ja konkrētā tvērumā definēti makro tiktu mantoti līdzīgi ka mainīgie, kas nozīmē, ka šabloni, kas ir specifiski tvērumam, būtu ar lielāku prioritāti ka tie, kas definēti vispārīgākā tvērumā.

FIXME: *Ko var un ko nevar salīdzinājumā ar C priekšprocesoru?*

6. Prototipa realizācija

Lai ilustrētu šādas sistēmas izstrādes iespējamību, tika izstrādāts prototips, kas parāda, 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 atklājams uz jebkuras mašīnas ar uzstādītu 2.7.0 Python versiju.

Šīs nodaļas apakšnodaļa 6.4. savukārt apraksta problēmas ar kurām saskārās darba autors un izņēmumus, kas pagaidām netiek implementēti prototipā.

FIXME: *pārrakstīt atkarībā no satura*

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 6.3. 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 izteiksme `t_id('f')` sagaidīs tieši identifikatoru `f`, bet izteiksme `t_id()` sagaidīs jebkuru identifikatoru. Šie tokeni nav reāli, to klases tika izvēlētas uzskatāmības 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 neattiecas 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 šis 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.

Viena no galvenām šīs sistēmas īpašībām ir iespēja atšķirt programmatūras kontekstus. Ja sistēma darba gaitā sastapās ar konteksta sākuma simbolu, tā izveido eksistējošā automāta kopiju un tālāk konteksta makro pievieno šai kopijai. Konteksta iekšienē strādā tādi paši likumi par izteiksmju prioritātēm - izteiksme, kas bija agrāk ir ar lielāku prioritāti. Bet makro, kas ir specifiski kontekstam ir ar lielāku prioritāti nekā vispārīgāki makro. Tātad jā pēc kārtas atnāks $t_id() ()$, konteksta sākuma tokens un $t_id() (t_real() *)$, tad otrā konteksta ietvaros virkne $t_id('f') ()$ tiks akceptēta ar otro regulāro izteiksmi. Pēc izejas no konteksta konteks-spezifiskais automāts tiek izmests ārā un darbs tiek turpināts ar automātu no iepriekšējā kontekstu līmeņa.

Konteksti ir implementēti tieši šādi, jo sapludinātā automāta atbrīvošana no vairs nevajadzīgiem pamestā konteksta stāvokļiem ir diezgan darbietilpīgs uzdevums. Ja n ir konteksta regulāro izteiksmju daudzums un m - maksimālais nesapludinātā automāta garums, tad dzēšanai būs vajadzīgs vismaz $O(n * m)$ laiks.

Sistēma turpina darbu aprakstītā gaitā līdz ieejas tokenu virknes beigām. Pēc sistēmas apstāšanās tiek iegūta jauna tokenu virkne, kas tika apstrādāta attiecīgi kodā ierakstītiem makro. Kad sistēma tiks integrēta ar reālu kompilatoru, tā strādās paralēli ar parsētāju un sistēmas izejas tokenu virkne tiks apstrādāta ar standartiem valodas likumiem.

6.2. Lietotie algoritmi

6.2.1. Determinizācija

FIXME: *Write!*

6.2.2. Minimizācija

FIXME: *Write!*

6.2.3. Apvienošana

FIXME: *Write!*

6.3. Realizācijas pamatojums

FIXME: *Write!*

Tika izvēlēta tieši šā

Kāpēc sapludina visu kopā, kāpēc minimizē,

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.

6.4. Problēmas un izņēmumi

FIXME: *Write!*

6.4.1. Problēmas

Problēmas?

6.4.2. Izņēmumi

Prototipā pagaidām nav implementēta apstrādes dalīšana pa gramatikas produkcijām, visas regulārās izteiksmes ir sapludinātas vienā automātā. Regulāro izteiksmju dalīšana pa tipiem tiks izstrādāta vēlāk, kad tiks uzsākta integrācija un sadarbība ar reālu kompilatoru. Tā varētu tikt implementēta līdzīgi tam, kā tiek realizēti konteksti - pa vienam sapludinātam automātam priekš katra produkcijas tipa.

Sistēma nezina neko par valodas gramatiku. Tieši tāpēc tokens `{real}` netiks uzverti ka `{expr}`, kaut arī racionāls skaitlis ir izteiksme. Tā kā sistēmai jābūt neatkarīgai no valodas gramatikas, šī hierarhija nav iekodējama transformāciju sistēmā. To ir jānodrošina parsētājam, attiecīgi apstrādājot tokenus un apkopojot to nozīmi. Par to arī būs jā rūpējas programmētājam rakstot savas makro izteiksmes.

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?*

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. Why adaptable grammars are cool

Adaptīvās gramatikas ir ļoti spēcīgs rīks kompilatoru un parsētāju būvēšanā. Static semantics - gandrīz sintakse, bet ne gluži

9.3. Why adaptable grammars suck

В общем к ним сложно написать формализм и они не особо применимы реально, потому что очень уж ограничены.

Адаптирующиеся грамматики в общем случае мало того, что требуют пересчитывания всей таблицы парсинга - омиг, так ещё и могут накосячить с тем, что потеряется парсабельность самой грамматики - LL(1) или LALR() или ещё какая-нибудь, которая необходима для того, чтобы работал парсер. Следовательно фиг ты их прикрутишь в обычном виде к уже существующему парсеру [2]

Literatūra

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [2] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25(11):35--44, November 1990.
- [3] 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.
- [4] Forbes D. Lewis. Recursive descent parsers. <http://www.cs.engr.uky.edu/~lewis/essays/compilers/rec-des.html>.
- [5] John N. Shutt. Adaptive grammars. <http://web.cs.wpi.edu/jshutt/>, November 2005.