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

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

1.1. Regulārās izteiksmes

FIXME: *Uzrakstīt!* Regulārās izteiksmes, kas tie ir un ko ar tām var darīt.

1.2. Priekšprocesori

FIXME: *Uzrakstīt!* Kas tie ir un to iespējas.

1.3. Programmas tvērumi

Programmas tvērums ir programmas bloks, kurā definēti mainīgo nosaukumi vai citi identifikatori ir lietojami, un kurā to definīcijas ir spēkā. Programmas ietvaros tvērumus 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.

Tvēruma 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. Tvēruma iekšā tiks samainīta globālā mainīgā `b` vērtība, jo tas netika pārdefinēts.

1.4. Type inference

FIXME: *Uzrakstīt!*

2. Ievads

Kontekstneatkarīgas gramatikas ir vienkāršs un uzskatāms rīks, kas ļauj aprakstīt programmēšanas valodu sintaksi. Diemžēl, mūsdienīgo programmēšanas valodu sintaksi ir grūti aprakstīt ar viennozīmīgām kontekstneatkarīgām gramatikām. Vairākumam modernu valodu parsētāji ir rakstīti ar rokām, programmētājam uzmanīgi risinot gramatikas konfliktus. Eksistē arī parsētāju ģeneratori, piemēram ANTLR, kas ar ieskatu uz k tokeniem uz priekšu tokenu virknē var izlemt, kā atrisināt parsēšanas neviennozīmības. Tomēr lietojot šādus parsētāju ģeneratorus ir jāprot veidot tiem saprotamas gramatikas un konfliktu risināšanas noteikumus.

Šī darba mērķis ir aprakstīt iespēju dot programmētājam papildināt programmēšanas valodu sintaksi. Lai to īstenot, var piedāvāt divus variantus. Pirmais no tiem ir pārrakstīt vai pārģenerēt parsētāju līdzko parādās nepieciešamība ieviest jaunas konstrukcijas. Ir pašsaprotami, ka radikālu gramatikas izmaiņu gadījumā tas tiešām būs nepieciešams. Tomēr gadījumos, kad izmaiņas tiek veiktas lai atvieglotu programmētāja darbu vai padarītu programmas kodu pārskatāmu, parsētāja pārrakstīšana ir lieks darbs. Otrais variants, pie kura pieturas dotais darbs, ir iznest sintakses modificēšanu uz lietotāja līmeni. Tas nozīmē, ka valodas lietotājam jāpiedāvā kaut kāds mehānisms valodas konstrukciju manipulēšanai.

Dotais darbs apskata sistēmu, kas ļauj dinamiski paplašināt programmēšanas valodu sintaksi. Par pamata principu šīs sistēmas izstrādē ir ņemts pašmodificējamo gramatiku jēdziens. Par dinamiski modificējamo gramatiku iespējamību un realizējamību iet runa dažādos rakstos. Tomēr vispārīgā gadījumā šādām gramatikām trūkst jebkādu kontroles mehānismu. Tādas gramatikas var nekontrolējami mainīties, aizvietojot sākotnējo gramatiku ar pavisam jaunu. Tas var izraisīt dažādas problēmas, bet galvenās no tām ir parsējamības bojāšana (piemēram, kreisās rekursijas ieviešana LL parsētāju gadījumā) un middle-end savietojamības zaudēšana (kompilators vairs var nesaprast parsētāja izveidoto sintakses koku).

Piedāvātā sistēma ir balstīta uz sakarīgu gramatikas modifikāciju iespēju ierobežojumu, kā arī uz tipu sistēmas, kas ļaus pārliecināties, ka ieviestās modifikācijas ir korektas. Tas tiks nodrošināts ar specifisku makro šablonu sintaksi. Sistēmas galvenā īpašība ir tas, ka pēc gramatikas transformācijas izpildes modificētais izejas kods būs atpazīstams ar sākotnējo valodas gramatiku.

Aprakstāmā sistēma nav piesaistīta pie kādas konkrētas programmēšanas valodas. Tā tiek izstrādāta bāzējoties uz LL parsētāju ar noteiktām īpašībām. Perspektīvā tā varēs tikt lietota jebkurai valodai, kuras parsētājam piemīt līdzīgas īpašības un piedāvāt šai valodai pašmodificēšanas iespējas. Lietojot makro šablonus sakrišanas atrašanai kodā un vienkāršu programmēšanas 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 tokeniem. Prototipa izstrādes galvenais uzdevums ir atrast efektīvu veidu, ka 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 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. 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. Uzdevuma pamatojums

3.1. Par programmēšanas valodu reprezentāciju

Programmēšanas valodas ir jēdzienu sistēma, kas ļauj aprakstīt algoritmus. Šai sistēmai jābūt saprotamai programmētājam, tāpēc tiek meklēti veidi, kā tās sintaksi var nodefinēt formāli. Vienkāršs un intuitīvi saprotams rīks, kas der šīm uzdevumam, ir kontekstneatkarīgas gramatikas.

Kontekstneatkarīgas gramatikas piedāvāja N. Homskis, kas plānoja lietot tos lai ieveidotu reālo cilvēku valodu modeļus. Šinī jomā tās gandrīz netiek lietotas, jo dabiskās valodas ir pārāk sarežģītas un ar daudziem izņēmumiem no gramatikas likumiem. Tomēr šīs gramatikas tiek lietotas lai vispārināti aprakstītu programmēšanas valodu sintaksi. Programmēšanas valodas globālā līmenī nav kontekst-neatkarīgas, bet tomēr tās ir neatkarīgas lokāli. Kaut arī ne visas programmēšanas valodu īpašības var aprakstīt ar kontekstneatkarīgām gramatikām, tās ir ērti lietot lai parādīt valodas konstrukciju struktūru.

Kontekstneatkarīgas gramatikas sastāv no četrām daļām. Pirmā ir simbolu kopa, kas tiek saukta par termināliem simboliem. Otrā ir simbolu kopa, kas tiek saukta par neterminālie simboliem. Trešā ir gramatikas sākuma simbols, kas ir viens no neterminālu simbolu kopas. Un, beidzot, ceturtā gramatikas daļa ir pārrakstīšanas likumu kopa. Termināle simboli ir gramatikas definētās valodas vārdnīca. No tiem tiks sastādīta valoda, kuru definē dotā gramatika. Neterminālie simboli, savukārt, var tikt apskatīti ka terminālu un terminālu virkņu klases.

Pārrakstīšanas likumi tiek pierakstīti izskatā $\mathtt{A} \rightarrow \mathtt{b}$, kur $\mathtt{A}$ ir viens no neterminālie 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. $\mathtt{A} \underset{G}{\overset{*}{\Rightarrow}} \mathtt{b}$ parāda, ka $\mathtt{A}$ var tikt pārveidots virknē $\mathtt{b}$ lietojot gramatikas G pārrakstīšanas likumus. Šādas secīgu pārveidojumu rinda tiek saukta par atvasinājumu. [3]

Pārveidojumu rindas var tikt attēlotas koku veidā. Šīs koks skaidri parāda kā simboli no terminālu virknes tiek grupēti apakš-virknēs, katra no kurām pieder kādam no neterminālie simboliem. Bet vēl svarīgāk, šīs koks, ko sauc par parsēšanas koku, ir struktūra, kas reprezentē apstrādājamo programmu. Kompilatorā šāda struktūra veicina programmas izejas teksta translāciju uz izpildāmu kodu. Gadījumā, ja gramatikā eksistē divi parsēšanas koki vienam un tam pašam atvasinājumam, gramatika ir neviennozīmīga. Neviennozīmības padara gramatikas nelietojamas programmēšanas valodu aprakstam, jo šādā gadījumā kompilators nevarēs izsekot pareizu programmas struktūru. [2]

Parsētāji ir programmas, kas izpilda programmas teksta pārstrādi parsēšanas kokā. Vairākums parsētāju mūsdienās aktuālākam valodām (piemēram C/C++) ir rakstīti manuāli. Parasti lietojamas parsēšanas pieejas var tikt sadalītas 2 grupās - top-down and bottom-up parsēšana. Abām pieejām ir ierobežotas gramatikas klases, ko tie prot atpazīt. Bet patiesībā šīs klases ir diezgan plašas, tāpēc ļauj aprakstīt vairākumu sintaktisko konstrukciju mūsdienīgām valodām.[1]

FIXME: *Parsētāju tipi - LR, LL, to īpašības*

Bet parsētāji nestrādā pa taisno ar programmas izejas tekstu, ko sastāda leksēmas. Parsētāji strādā ar jau iepriekš apstrādātu izejas tekstu, kas tika pārveidots tokenu virknē.

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 apzimēti 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 identifikators. Tokena vērtībā nonāk identifikatora nosaukums `sum`.

2. Piešķiršanas operators = tiks pārveidots tokenā `{=}` Šīm 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` analoģiski `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]

Šī darbā arī tiek lietots jēdziens pseido-tokens. Pseido-tokens ir citu tokenu grupa, kas tiek aizvietota ar vienu objektu. Tas var tikt darīts, lai vienreiz noparsētu izteiksmi nevajadzētu apstrādāt vēlreiz. Tokenu aizvietošana ar pseido-tokeniem notiek gramatikas likumu reducēšanas brīdī. Kad, piemēram, tokenu virkne `{id:a} '+' {id:b}` tiek atpazīta ka derīga izteiksme gramatikas ietvaros, tā var tikt aizvietota ar pseido-tokenu `{expr:a + b}`.

3.2. Par dinamiskām gramatikām

Starp īpašībām, kuras nevar aprakstīt ar kontekstneatkarīgām gramatikām ir leksiskais tvērums ($lexicalscope$) un statiskā tipizācija ($statictyping$).

FIXME: *Uzrakstīt!* Dinamiskas vai adaptīvās gramatikas ir gramatiskais formālisms, kas ļauj modificēt gramatikas likumu kopu ar gramatikas rīkiem. [3]

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 tās daudz nepētīja un daudz reāli nelieto.* Tās vispārīgā gadījumā ir nekontrolējamas.

3.2.1. Kā tos parasti vēlās lietot

adding grammar rules when adding a variable - makes static semantics easier to control.
BUT: problems with scope, recursion and other stuff [**?**]

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 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.

FIXME: *Continue*

5. Transformāciju sistēma

Ka var redzēt no nodaļas 3.2., 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 pilnībā aizvietota ar citu. 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.

Vēl viena problēma adaptīvo gramatiku lietošanā ir tas, ka tā nevar tikt pielietotas valodām, kurām jau eksistē kompilatori, bez attiecīgas parsētāju modifikācijas. Bet tā kā mūsdienīgo valodu gramatikas ir diezgan sarežģītas, parsētāju pārrakstīšana un dinamisko izmaiņu iespējas pievienošana var kļūt par lieku spēku tērēšanu. Visērtākais veids, kā ļaut programmētājam modificēt valodas gramatiku ir izveidot papildus sistēmu, kas to varēs nodrošināt ar minimālām izmaiņām jau eksistējošā parsētājā.

Šīs darbs piedāvā uzbūves principus sistēmai, kas tiek domāta ka palīgrīks parsētājam un kura dos iespēju programmētājam dinamiski paplašināt valodas iespējas ar makro valodas palīdzību. Šī makro valoda ļaus izveidot jaunas valodas konstrukcijas no jau eksistējošām vienībām. Tas tiks realizēts analizējot kodu ar ierakstītiem šabloniem un apstrādājot atrastās pseido-tokenu virknes, transformējot tos citās struktūrās, kas varēs tikt atpazītas ar sākotnējo valodas gramatiku. Apstrādes rezultāts - jauna pseido-tokenu virkne - aizvietos attiecīgu koda gabalu. Nekādas pavisam jaunas konstrukcijas 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.

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 gramatikas 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 Compiler Technology & Computer Architecture Group, University of Hertfordshire (Hertfordshire, England), Heriot-Watt University (Edinburgh, Scotland) un Moscow Institute of Physics and Technology (Dolgoprudny, Russia). Šīs valodas sintakse bāzējas uz LaTeX teksta procesora sintakses, kas ir standarts priekš zinātniskām publikācijām. Korekti uzrakstīta Eq valodas programma var tikt

interpretēta ar LaTeX procesoru. Perspektīvā Eq programma varēs tikt kompilēta un izpildīta uz vairākuma mūsdienīgo arhitektūru.

Lai atvieglotu izstrādi valodā Eq tika nolemts izveidot makro sistēmu, kas ļaus pielāgot sintaksi programmētāja vajadzībām. Tomēr bez kaut kādas šablonu sistēmas makro iespējas ir ļoti ierobežotas. Tāpēc tika izlemts lietot šablonus ar minimālu regulāro izteiksmju sintaksi, kas dod brīvību sakritību aprakstīšanai.

Lai izveidot jaunas konstrukcijas no tokeniem, kas tika atpazīti ir nepieciešami kaut kādi rīki, lai apstrādāt tokenus, kas tika atpazīti, ka sakrītoši ar vienu no šabloniem.

tāpēc tika izlemts lietot regulāro izteiksmju šablonus. kas dod brīvību sakrišanas meklēšanas mehānismam. Tālāk, lai apstrādāt regulārās izteiksmes sakrātos tokenus, tika nolemts izveidot vienkāršu funkcionālu valodu, kas ļaus pārstrādāt pseido-tokenu virknes atkarībā no programmētāja izveidotiem šabloniem.

Bet 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 parsētājam kas atbilst dažiem nosacījumiem. Par parsētājiem nepieciešamām īpašībām tiks runāts apakšnodaļā 5.2..

5.2. Parsētāji

Šajā darbā piedāvātā sistēma tiek izstrādāta uz LL(k) parsētāja bāzes. 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 pieejam, 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, kas samazina kļūdu varbūtību.

Tā kā transformāciju sistēma tiek veidota ka paplašinājums parsētājam, parsētājam jāatbilst dažiem nosacījumiem, kas ļaus sistēmai darboties. Zemāk ir 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 patvaļīgas vietas tokenu virknē.

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`).

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` → `(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)` → `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 pseido-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 izpilda transformācijas ar tokenu virknēm, kas tiek akceptētas ar makro kreisās puses šablonu.

Apskatīsim *match* funkciju likumus, kas modificē apstrādājamās gramatikas produkcijas uzvedību. *Match* makro sintakses vispārīgo formu var redzēt figūrā 1..

```
match [\prod1] v = regexp → [\prod2] f(v)
```

1. att. *Match* makro sintakses vispārīgā forma

Šīs apraksts 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-regexp | regexp
concat-regexp   → asterisk-regexp  concat-regexp
asterisk-regexp → unary-regexp * | unary-regexp
unary-regexp    → 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 - `|{expr}|`. Sākotnējā valodas gramatikā eksistē absolūtās vērtības funkcija izskatā `abs({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. Transformācijas pieeja

Šī nodaļa satur aprakstu par to, kā tiek plānots izveidot programmētājam saprotamu transformēšanas mehānismu un kontrolēt to iespējas.

Ir nepieciešams izveidot mehānismu, kas ļaus transformēt makro kreisās puses akceptētu pseido-tokenu virkni, izveidojot virkni, kas to aizvietos. Lai to izdarītu ir nepieciešama kaut kāda programmēšanas valoda, par kuru ies runa apakšnodaļā 5.4.1..

Ir plānots, ka transformāciju sistēma varēs atpazīt nepareizi sastādītus makro šablonus lietojot tipu kontroles pieeju. Šīs pieejas bāzes principi ir aprakstīti apakšnodaļā **??**.. Jāņem vērā tas, ka lai šī sistēma varētu tikt pielietota, izvēlētai transformāciju valodai jāpiemīt tipu secināšanas (*type inference*) īpašībai.

5.4.1. Tokenu virkņu apstrāde

Lai varētu izpildīt atrastās tokenu virknes apstrādi un modificēšanu ir nepieciešams kaut kāds papildus rīks. Šīs rīks varētu būt kaut kāda programmēšanas valoda. Šādai pieejai ir divas iespējas - imperatīvā valoda vai funkcionālā valoda.

Šīm uzdevumam varētu lietot kādu no imperatīvam programmēšanas valodām, piemēram C, vienkārši izveidojot saskarni ar tās valodas kompilatoru. Bet vairākumam šādu valodu nav tipu secināšanas iespējas. Tipu secināšana C valodas gadījumā arī ir apgrūtināta ar rādītāju mainīgiem, kuru tipus nevar droši izrēķināt parsēšanas laikā. Lai varētu ieviest stingrās tipu izsecināšanas iespējas, vajadzēs ierobežot valodas iespējas, tātad modificēt eksistējošo kompilatoru vai kaut kā citādāk ierobežot pieejamo konstrukciju kopu.

Varētu lietot arī vienu no jau eksistējošām funkcionālām valodām ar tipu secināšanas īpašību, kas piemīt vairākumam funkcionālo valodu. Tomēr arī funkcionālām valodām ir daudz iezīmju, kas nav nepieciešami dotā uzdevuma risināšanai. Piemēram, slinkā rēķināšana šajā gadījumā nav nepieciešama, jo programmas izpildes laikā visas vērtības jau būs zināmas un slinkie aprēķini nebūs vajadzīgi. Vēl viena ērtā funkcionālo valodu īpašība ir tas, ka tās funkcijām nepiemīt blakusefekti, tātad to izpilde nevarēs samainīt eksistējošos datus. Valoda, kuras funkcijām ir blakusefekti, varētu sabojāta parsētāja darbu.

Šīs sistēmas implementācijā tika izvēlēts papildus izveidot vienkāršu funkcionālu valodu, kura būs statiski tipizējama. Tātad visiem šīs valodas mainīgajiem varēs izsecināt piederību pie tipa un pie kaut kāda virstipa, kas tiks lietots lai nodrošināt transformāciju korektību.

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 ka `cons(1, cons(2, cons(3, nil)))`, kur `nil` ir speciāls tukšs objekts. Valoda saturēs arī `if` konstrukciju, kas ļaus pārbaudīt dažādus nosacījumus.

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.

Tiek plānots, ka šī valoda arī ļaus izpildīt daļēju novērtējumu izteiksmēm, tur kur būs nepieciešams. Tas nozīmē, ka valodai jāsatur 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}`. Valoda arī ļaus izveidot jaunus tokenus ar izrēķinātu vērtību.

Funkcija `type`, savukārt, ļaus pārbaudīt tokenu tipu, kas var būt nepieciešams transformā-

cijas procesā, piemēram, lai atpazīt kādu operatoru.

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.

5.4.2. Tipu sistēma

Kā bija redzams figūrā 1., katrā makro pusē ir atrodams produkcijas nosaukums, `[prod1]` un `[prod2]`. Tas tiek darīts tādēļ, lai kontrolētu, kad dotais makro ir pārbaudīts, un kāda tipa izejas virkni tas radīs. Abas šīs atzīmes ir rādītas tipu kontroles sistēmas dēļ.

Katrs atsevišķs makro strādā konkrētas gramatikas produkcijas ietvaros, `[prod1]` dotā makro gadījumā. Tas nodrošinās to, ka katrs no makro tiks izpildīts pareizajā vietā un visas konstrukcijas tiks apstrādātas.

Otrais tips, `[prod2]`, atzīmē to, ka pēc transformācijas procesa beigām mums jāsaņem tieši šādai produkcijai korektu izteiksmi. Tātad ir jāparbauda tas, ka funkcijas `f(v)` rezultāts attiecībā uz atrasto tokenu virkni, ir atļauta ieejas virkne priekš produkcijas `prod2`.

Lai to paveikt ir nepieciešams izveidot pseido-tokenu regulāro izteiksmi produkcijai `prod2`. Tālāk ir nepieciešams izsecināt funkcijas `f` no virknes `v` rezultāta tipu.

Šajā darbā netiks apskatīts jautājums, kādā veidā tiks izveidota regulārā izteiksme priekš katras gramatikas produkcijas. To varētu izveidot programmētājs, vai, varbūt tā varētu tikt izveidota automātiski. Ir svarīgi pieminēt, ka pāreja no gramatikas likuma uz regulāro izteiksmi noved pie kādas informācijas zaudēšanas. Piemēram, nav iespējams uzkonstruēt precīzu regulāro izteiksmi valodai:

```
A :=  aAb | ab
```

Tomēr ir iespējams izveidot regulāro izteiksmi kas iekļaus sevī gramatikas aprakstīto valodu, piemēram, `a+b+`. Makro lietotā transformācijas shēma tiks atzīta par pareizo, ja ir iespējams pierādīt, ka produkciju aprakstošā regulārā izteiksme atpazīst arī valodu, ko veido `f(v)`.

Ir viegli pamanīt, ka regulārās izteiksmes izveido dabisku tipu hierarhiju. Valoda, kura var tikt atpazīta ar regulāro izteiksmi r_1, var tikt iekļauta citas regulārās izteiksmes r_2 atpazītās valodā apakškopas veidā. Piemēram, regulārās izteiksmes `a+` valoda ir atpazīstama arī ar regulāro izteiksmi `a*`, bet `a*` atpazīst vēl papildus tukšu simbolu virkni. Šādai tipu hierarhijai uz regulārām izteiksmēm eksistē arī super-tips, ko uzdod regulārā izteiksme `.*` - $\top$. Ir acīmredzami, ka $\forall t_i \in R, t_i \sqsubseteq \top$, kur R ir visu regulāro izteiksmju kopa.

Ir svarīgi izveidot procedūru, kas ļaus izsecināt, vai $r_1 \sqsubseteq r_2$. Ir zināms, ka ir iespējams katrai regulārai izteiksmei uzbūvēt minimālu akceptējošu galīgu determinētu automātu. Šīs automāts atpazīs precīzi to pašu valodu, ko atpazīst regulārā izteiksme. Tas nozīmē, ka no $r_1 \sqsubseteq r_2 \Rightarrow sekomin(det(r_1)) \sqsubseteq min(det(r_2))$. Diviem minimāliem automātiem A_1 un A_2, $A_1 \sqsubseteq A_2$ nozīmē, ka eksistē kaut kāds attēlojums Ψ no A_1 stāvokļiem uz A_2 stāvokļiem, tāds, ka:

$$Start(A_1) \rightarrow Start(A_2) \in \Psi$$

$$\forall s \in States(A_1) \forall e \in Edges(s), \Psi(Transition(s, e)) = Transition(\Psi(s), e)$$

Šeit $States(x)$ apzīmē automāta x stāvokļu kopu, $Edges(s)$ apzīmē pseido-tokenu kopu, kas atzīmē no stāvoļa s izejošās šķautnes. $Transition(s, t)$, savukārt, apzīmē stāvokli, kas ir sasniedzams no s pārejot pa šķautni, kas atzīmēta ar pseido-tokenu t.

Otra svarīga īpašība, kas tiks lietota šajā tipu pārbaudīšanas sistēmā ir tas, ka transformāciju valoda ir statiski tipizējama un tā satur ļoti ierobežotu iebūvēto funkciju skaitu. Katrai no šīm iebūvētām funkcijām ir iespējams izveidot to aprakstošo regulāro izteiksmi. Piemēram, regulārā izteiksme funkcijai $head(x)$ var tikt izveidota ka visu to šķautņu kopa, kas iziet no x aprakstošā automāta sākuma stāvokļa.

Var redzēt, ka šāda tipu pārbaudīšanas sistēma tik tiešam ir teorētiski iespējama. Sīkāka informācija par tipu sistēmu ir saņemama pie Eq kompilatora izstrādes komandas.

5.5. Sistēmas sakars ar priekšprocesoriem

Ir dažādas pieejas programmu pirmkoda priekšprocesēšanai. Visvairāk izplatītas no tām ir divas pieejas. Viena no pieejām ir sintaktiskā pieeja - 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 līdz sistēmas darba izpildei parsētājs nevar uzbūvēt sintaktisko koku. Otra no pieejām ir leksiskā, leksiskie priekšprocesori tiek palaisti pirms pirmkoda parsēšanas un nezin neko par apstrādājamas valodas sintaksi (piem. C/C++ priekšprocesors).

Leksiskie priekšprocesori pēc savām īpašībām ir tuvi aprakstāmai sistēmai. Ar makro valodu palīdzību tiem tiek uzdoti koda pārrakstīšanas likumi, un kods tiek pārveidots attiecīgi tām. Bet leksisko priekšprocesoru vislielākais trūkums ir tas, ka tie apstrādā tekstu pa simboliem neievērojot izteiksmju un konstrukciju struktūru. Piemēram, apskatīsim 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 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.

Tā kā aprakstāmā sistēma strādās ar tokeniem un pseido-tokeniem, nevis ar tekstu, ar šādu problēmu tā nesastapsies. Konstrukciju `(a|b)+c` lekseris atpazīs ka pseido-tokenu `{expr}`, un sistēmas darbības laikā apstrādājamā plūsmā būs tieši pseido-tokens `expr`. Sistēmas darbošanās konkrētās produkcijas ietvaros arī atbrīvo sistēmu no nepieciešamības iekļaut zināšanas par valodas gramatiku, un it īpaši par pseido-tokenu īpašību mantošanas mehānismiem (piemēram, `{int}` arī ir `{expr}`, bet par šo transformāciju rūpēsies parsētājs).

Otrā šāda tipa priekšprocesoru problēma ir tas, ka tie strādā ārpus programmas tvērumiem. Tas nozīmē, ka tvēruma sākuma tokens (piemeram, `{` C/C++, 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.

Sistēmas sakrišanas meklēšanas mehānisms tiks izstrādāts ņemot vērā programmas tvēruma maiņu. Tātad šabloni, kuri tiek ieviesti konkrētā tvērumā, strādās tikai tā ietvaros. Sakrišanu meklēšanas mehānisma tvērumu realizācija tiks aprakstīta nodaļā 6.1..

6. Prototipa realizācija

Lai ilustrētu šādas transformāciju sistēmas izstrādes iespējamību, tika izstrādāts sakrišanu meklēšanas mehānisma prototips. 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.

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.

Prototips imitē darbu reālajā vidē, saņemot pa vienam tokenus no ieejas plūsmas no klases, kas imitē leksera darbu. 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 divas 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 divas izteiksmes `{id} '(' ')'` un `{id} '(' ({real}*) ')'`, tad ielasot virkni `{id:foo} '(' ')'` 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.

Sistēma arī ļauj veidot regulārās izteiksmes ar specifiskām tokenu vērtībām. Piemēram, regulārā izteiksme `{id:foo}` sagaidīs tieši identifikatoru `foo`, bet izteksme `{id}` sagaidīs jebkuru identifikatoru.

Prototips strādā pēc *greedy* principa - tas akceptē visgarāko iespējamo šablona sakritību. Ja eksistē divi šabloni, kas dod sakritību ar vienādu garumu, tad tiek ņemtas vērā prioritātes. Tas var notikt, ja eksistē divi šabloni, viens no kuriem satur vērtības prasību tokenam, bet otrs - nesatur. Ja tiek ielasītas divas izteiksmes `{id:foo} '(' {real}* ')'` un `{id} '(' {real}* ')'`, tad virkne `{id:foo} '(' {real:4.5} ')'` derēs abiem šabloniem. Bet tā kā šablons `{id:foo} '(' {real}* ')'` tika ielasīts pirmais, tieši tas arī būs akceptēts.

Viena no galvenām šīs sistēmas īpašībām ir iespēja atšķirt programmatūras tvērumus.

Ja sistēma darba gaitā sastapās ar konteksta sākuma simbolu, tā izveido eksistējošā automāta kopiju un tālāk tvēruma makro pievieno šai kopijai. Tvēruma iekšienē strādā tādi paši likumi par izteiksmju prioritātēm - izteiksme, kas bija agrāk ir ar lielāku prioritāti. Bet makro, kas ir specifiski tvērumam ir ar lielāku prioritāti nekā vispārīgāki makro. Tātad jā pēc kārtas atnāks `{id} '(' ')'`, tvēruma sākuma tokens un `{id} '(' {real}* ')'` , tad otrā tvēruma ietvaros virkne `{id:foo} '(' ')'` tiks akceptēta ar otro regulāro izteiksmi. Pēc izejas no tvēruma tā specifiskais automāts tiek izmests ārā un darbs tiek turpināts ar automātu no iepriekšējā tvēruma līmeņa.

Tvērumu pārvalde ir implementēta šādā veidā, jo sapludinātā automāta atbrīvošana no vairs nevajadzīgiem pamestā tvēruma stāvokļiem ir diezgan darbietilpīgs uzdevums. Ja n ir tvēruma regulāro izteiksmju daudzums un m - maksimālais nesapludinātā automāta garums, tad dzēšanai būs vajadzīgs vismaz $O(n * m)$ laiks.

Tad, kad atnāk kaut kādi tokeni, kas neatzīmē makro sākšanās, sistēma izpilda pārejas starp sapludinātā automāta stāvokļiem un atceras tokenus, kurus jau ir nolasījusi. Sistēma atrod garāko virkni, kas atbilst kādam no šabloniem un tad atgriež tās identifikatoru un nolasīto tokenu virkni, lai turpmāk transformēšanas mehānisms varētu pārstrādāt to jaunajā virknē.

Pieņemot, ka transformēšanas sistēma ir izstrādāta, tālākā darba gaita būs sekojoša. Transformēšanas sistēma aizstāv ielasīto virkni ar citu, kas ir konstruēta pēc akceptētās regulārās izteiksmes noteikumiem. Tad sakrišanas meklēšanas sistēmas darbs tiek uzsākts no aizvietotās virknes sākuma.

Sistēma turpina darbu aprakstītā gaitā līdz ko neviens no šabloniem vairs netiek akceptēts. 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: *Uzrakstīt!*

6.2.2. Minimizācija

FIXME: *Uzrakstīt!*

6.2.3. Apvienošana

FIXME: *Uzrakstīt!*

6.3. Realizācijas pamatojums

FIXME: *Uzrakstīt!*

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. Izņēmumi

FIXME: *Papildināt*

6.4.1. Transformācijas

Šīs prototips nenodarbojas ar tokenu virkņu transformācijām, jo tā nav sakrišanu meklēšanas mehānisma

6.4.2. Produkcijas

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 tam, kā tiek realizēti konteksti - pa vienam sapludinātam automātam priekš katra produkcijas tipa.

6.4.3. Tokenu mantošana

Sistēma nezin neko par valodas gramatiku. Tieši tāpēc tokens `{real}` netiks uztverts 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 jārūpējas programmētājam rakstot savas makro izteiksmes.

6.4.4. Tokenu vērtības

Vienlaikus `{id:f}` un `{id}` ievieš nedeterminētību, kaut arī visi automāti sistēmā ir determinizēti.

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).

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] 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.

[3] John N. Shutt. Recursive adaptive grammars. Master's thesis, Worcester Polytechnic Institute, August 1993.