

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.	Ievads . . . . .	5
2.	Uzdevuma pamatojums . . . . .	7
2.1.	Par programmēšanas valodu reprezentāciju . . . . .	7
2.2.	Par dinamiskām gramatikām . . . . .	9
2.2.1.	Kā tos parasti vēlās lietot . . . . .	9
3.	Transformāciju sistēma . . . . .	10
3.1.	Idejas rašanās - valoda Eq . . . . .	10
3.2.	Parsētāji . . . . .	11
3.3.	Makro sistēmas sintakse . . . . .	12
3.4.	Transformācijas pieeja . . . . .	13
3.4.1.	Tokenu virkņu apstrāde . . . . .	14
3.4.2.	Tipu sistēma . . . . .	15
3.5.	Sistēmas sakars ar priekšprocesoriem . . . . .	16
4.	Prototipa realizācija . . . . .	18
4.1.	Atļautā makro sintakse . . . . .	18
4.2.	Vispārīgā pieeja . . . . .	18
4.3.	Makro konfliktu risināšana . . . . .	19
4.3.1.	Divu makro konflikts vienā tvērumā. . . . .	19
4.3.2.	Divu makro konflikts dažādos tvērumos. . . . .	19
4.3.3.	Dažādu virkņu garumu konflikts . . . . .	19
4.4.	Realizācijas pamatojums . . . . .	19
4.5.	Kāpēc neder jau uzrakstītas bibliotēkas . . . . .	20
4.6.	Lietotie algoritmi . . . . .	21
4.6.1.	Regulāro izteiksmju pārveidošana nedeterminētā galīgā automātā . . . . .	21
4.6.2.	Determinizācija . . . . .	21
4.6.3.	Minimizācija . . . . .	22
4.6.4.	Apvienošana . . . . .	22
4.6.5.	Sakrišanu meklēšana . . . . .	22
4.6.6.	Tvērumi . . . . .	22
4.7.	Izņēmumi . . . . .	22
4.7.1.	Transformācijas . . . . .	22

4.7.2.	Produkcijas . . . . .	22
4.7.3.	Tokenu mantošana . . . . .	23
4.7.4.	Regulārās izteiksmes daļu grupēšana . . . . .	23
4.7.5.	Tokenu vērtības . . . . .	23
4.8.	Optimizācijas iespējas . . . . .	23
5.	Rezultāti . . . . .	24
5.1.	Prototipa īpašības . . . . .	24
6.	Secinājumi . . . . .	24

## Termini un apzīmējumi

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

### Regulārās izteiksmes

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

### Atpakaļnorādes

**FIXME:** *Uzrakstīt!*

### Priekšprocesori

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

### 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 ar 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.

## **Type inference**

**FIXME:** *Uzrakstīt!*

## **Epsilon-pārejas**

**FIXME:** *Uzrakstīt!*

# 1. Ievads

Основная цель этой работы - показать, что возможно сделать грамматику динамически парсируемой, контролируя её модификации. Данная работа представляет теоретическую базу для системы, которая позволит на лету расширять синтаксис языка. Представленная система является надстройкой над парсером, следовательно для её использования нет необходимости вносить значительные изменения в его работу.

Система разрабатывается в составе группы студентов университетов разных стран. Автор данной работы участвовал в разработке идеи в целом, однако фокусом была разработка подсистемы, которая позволит находить появления совпадения с шаблонами в коде программы.

В основе предложенной системы лежит принцип динамических - самомодифицирующихся - грамматик. Проблема с динамическими грамматиками заключается в том, что они могут много всего и контролировать их очень сложно. Однако данная работа выдвигает некоторые ограничения для контроля корректности изменений, вносящихся в грамматику. Для этого будет использоваться система вывода типов, при помощи которой можно будет следить за тем, чтобы переписывалось нужное и в нужное.

Данная система не позволит добавлять и удалять правила грамматики на усмотрение программиста. Однако она даёт удобную возможность дополнять существующие конструкции новыми при помощи языка макросов. Это существенно упростит работу с языком, так как чтобы добавить удобные конструкции не удет необходимости исправлять парсер языка.

Чтобы расширить возможности языка макросов, его система шаблонов будет расширена регулярными выражениями. Одна из особенностей описываемой системы заключается в том, что она работает не на тексте, а на токенах, за счёт чего и возможно устроить систему вывода типов. Далее список заматченных токенов из текста программы будет преобразовываться соответственно правилу трансформации, описанному в макросе.

В рамках данной работы будет описана модель предлагаемой системы. На данный момент система находится в стадии прототипирования, поэтому типовая система и система трансформаций будет только обозначена, а не рассмотрена полностью. Эта работа фокусируется на разработке эффективного прототипа для системы поиска совпадений в коде.

На момент написания работы есть готовый прототип системы поиска совпадений с шаблонами макросов. Основной принцип действия прототипа - создание одного детерминированного автомата для поиска совпадений по всем шаблонам по одному проходу по коду, по возможности за линейное время.

Далее организация работы будет следующей. Глава 2 рассказывает об актуальности проблемы, об общепринятом подходе к написанию парсеров языков и о возможностях динамических грамматик. Она также идентифицирует проблемы и сложности в использовании принципа самомодификации языка. Глава 3 рассказывает об основных принципах предлагаемой системы. Она даёт общий обзор языка модификаций и типовой системы, а также рассказывает о работе системы в рамках парсера языка. Глава 4 представляет прототип системы поиска совпадений с шаблонами, рассказывает об используемых алгоритмах и описывает причины выбора подходов. Там же описан подход к тестированию прототипа. В главе 5 описаны



результаты данной работы, глава 6 представляет выводы.

## 2. Uzdevuma pamatojums

### 2.1. 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 šim uzdevumam, ir kontekstneatkarīgas gramatikas.

Kontekstneatkarīgas gramatikas piedāvāja N. Homskis, kas plānoja lietot tos lai ievieidotu 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 netermināliem simboliem. Trešā ir gramatikas sākuma simbols, kas ir viens no neterminālu simbolu kopas. Un, beidzot, ceturrtā gramatikas daļa ir pārrakstīšanas likumu kopa. Terminālie simboli ir gramatikas definētās valodas vārdnīca. No tiem tiks sastādīta valoda, kuru definē dotā gramatika. Neterminālie simboli, savukārt, var tikt apskatīti ka termināļu un termināļu virkņu klases.

Pārrakstīšanas likumi tiek pierakstīti 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 aizvietojoit kreisēs puses netermināli ar labo likuma daļu.  $A \xRightarrow[G]{*} b$  parāda, ka  $A$  var tikt pārveidots virknē  $b$  lietojot gramatikas  $G$  pārrakstīšanas likumus. Šādas secīgu pārveidojumu rinda tiek saukta par atvasinājumu. [6]

Pārveidojumu rindas var tikt attēlotas koku veidā. Šīs koks skaidri parāda kā simboli no termināļu virknes tiek grupēti apakš-virknēs, katra no kurām pieder kādam no netermināliem simboliem. Bet vēl svarīgāk, šis 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. [5]

**FIXME:** manuāli bueee

Parsētāji ir programmas, kas izpilda programmas teksta pārstrādi parsēšanas kokā. Vairākums parsētāju mūsdienās aktuālākām 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.[2]

### **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 izejas 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 identifikators. 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}`. [2]

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}`.

## 2.2. Par dinamiskām gramatikām

Starp īpašībām, kuras nevar aprakstīt ar kontekstneatkarīgām gramatikām ir leksiskais tvērumš (*lexical scope*) un statiskā tipizācija (*static typing*).

**FIXME:** *Uzrakstīt!* Dinamiskas vai adaptīvās gramatikas ir gramatiskais formālisms, kas ļauj modificēt gramatikas likumu kopu ar gramatikas rīkiem. [6]

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.

### 2.2.1. 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 [3]

### 3. Transformāciju sistēma

Ka var redzēt no nodaļas 2.2., 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.

Šis darbs piedāvā uzbūves principus sistēmai, kas tiek domāta ka palīgriks 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.

#### 3.1. 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 L<sup>A</sup>T<sub>E</sub>X teksta procesora sintakses, kas ir standarts priekš zinātniskām publikācijām. Korekti uzrakstīta Eq valodas programma var tikt

interpretēta ar  $\text{\LaTeX}$  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 sakrīšanas meklēšanas mehānismam. Tālāk, lai apstrādāt regulārās izteiksmes sakrītos 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 3.2..

## 3.2. 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 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, 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 īstēnībā 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 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 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  $(\text{Parser}, \text{Production}) \rightarrow \text{Parser}$ . 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.

### 3.3. 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..

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

1. att. *Match* makro sintakses vispārīgā forma

Šis 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

### 3.4. 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 aizvieto. Lai to izdarītu ir nepieciešama kaut kāda programmēšanas valoda, par kuru ies runa apakšnodaļā 3.4.1..



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.

### 3.4.1. 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 vairākus 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 citādi ierobežot pieejamo konstrukciju kopu.

Varētu lietot arī vienu no jau eksistējošām funkcionālām valodām ar tipu secināšanas īpašību, kas piemīt vairākus 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 sabojāt parsētāja darbu.

Šīs sistēmas implementācijā tika izvēlēts papildus izveidot vienkāršu funkcionālu valodu, kura būs statistiski 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 kā `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 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}`. 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.

### 3.4.2. 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:

$$A := aAb \mid ab$$

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. Šis automāts atpazīs precīzi to pašu valodu, ko atpazīst regulārā izteiksme. Tas nozīmē, ka no  $r_1 \sqsubseteq r_2 \Rightarrow \text{sekomin}(\text{det}(r_1)) \sqsubseteq \text{min}(\text{det}(r_2))$ . Diviem minimāliem automātiem  $A_1$  un  $A_2$ ,  $A_1 \sqsubseteq A_2$  nozīmē, ka eksistē kaut kāds attēlojums  $\Psi$  no  $A_1$  stāvokļiem uz  $A_2$  stāvokļiem, tāds, ka:

$$\text{Start}(A_1) \rightarrow \text{Start}(A_2) \in \Psi$$

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

Šeit  $States(x)$  apzīmē automāta  $x$  stāvokļu kopu,  $Edges(s)$  apzīmē pseido-tokenu kopu, kas atzīmē no stāvokļa  $s$  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 statistiski 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 tiešām ir teorētiski iespējama. Sīkāka informācija par tipu sistēmu ir saņemama pie Eq kompilatora izstrādes komandas.

### 3.5. 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 nezina neko par apstrādājamās 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, 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 apstrādājamās 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 (piemēram,  $\{C/C++, Java \text{ un citu valodu gadījumā}\}$ ) 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 4.2..

## 4. Prototipa realizācija

**FIXME:** *This is just random text right now.*

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 atklājams uz jebkuras mašīnas ar uzstādītu 2.7.0 Python versiju.

Šīs nodaļas apakšnodaļa 4.7. savukārt apraksta problēmas ar kurām saskārās darba autors un izņēmumus, kas pagaidām netiek implementēti prototipā.

**FIXME:** *Pārrakstīt atkarībā no satura*

### 4.1. Atļautā makro sintakse

Kā jau bija pateikts, makro pieejamā regulāro izteiksmju sintakse ir minimāla. Tā atļauj lietot `*` lai identificēt tokenu virknes un `|` lai izvēlēties starp dažiem tokenu tipiem.

Prototips arī ļauj veidot regulārās izteiksmes ar specifisku tokenu vai pseido-tokenu vērtībām. Piemēram, regulārā izteiksme `{id:foo}` sagaidīs tieši identifikatoru `foo`, bet izteiksme `{id}` sagaidīs jebkuru identifikatoru.

### 4.2. Vispārīgā pieeja

Šī apakšnodaļa vispārīgi apraksta sistēmas prototipa darbību virkni.

Prototips imitē darbu reālajā vidē, saņemot pa vienam tokenus no ieejas plūsmas. Tokenu plūsmas saskarne imitē leksera darbu. Kamēr prototips nav saņēmis nevienu regulāro izteiksmi, tas ignorē tokenu plūsmas apstrādes izsaukumus. Tiklīdz prototipam tiek padota tokenu regulārā izteiksme, tas uzsāk regulārās izteiksmes parsēšanu. Parsēšanas procesā tiek izveidots galīgs automāts, kas akceptē regulārās izteiksmes uzdotās virknes.

Tad, kad atnāk tokenu apstrādes pieprasījums, sistēma izpilda pārejas starp automātu stāvokļiem, meklējot sakrišanas, 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.

### 4.3. Makro konfliktu risināšana

Makro šablonu konflikti var rasties tad, kad daži makro var tikt akceptēti vienlaikus. Tas var notikt gadījumos, ja divas regulārs izteiksmes akceptē līdzīgas virknes. Zemāk tiks aprakstīts, kā tika izvēlēts risināt dažādas konfliktu situācijas.

Reālajā situācijā var rasties 3 konfliktu veidi. Pirmais var rasties gadījumā, ja divas izteiksmes atpazīst vienu un to pašu virkni vienā tvērumā. Otrais var rasties gadījumā, ja jaunajā tvērumā parādās šablons, kas ir līdzīgs jau eksistējošam šablonam no vispārīgāka tvēruma. Trešais var rasties tad, kad viena no izteiksmēm akceptē kādu virkni, bet cita akceptē garāku virkni.

#### 4.3.1. Divu makro konflikts vienā tvērumā.

Gadījumā, ja viena tvēruma ietvaros eksistē divi šabloni, kas dod sakritību ar vienādu garumu, tad tiek ņemtas vērā prioritātes. 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.

#### 4.3.2. Divu makro konflikts dažādos tvērumos.

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, ja pirmajā tvērumā tiks ieviestas makro 1 un 2, bet otrajā tvērumā tiks ieviestas makro 3 un 4, to prioritāšu rinda izskatīsies sekojoši: 3, 4, 1, 2. Pirmie makro ir ar lielāku prioritāti, nekā tie, kas atnāca vēlāk, bet vēlāka tvēruma makro ir ar lielāku prioritāti nekā tie, kas atrodami agrākā tvērumā.

#### 4.3.3. Dažādu virkņu garumu konflikts

Prototips strādā pēc *greedy* principa - tas akceptē visgarāko iespējamo šablona sakritību. Tātad, ja eksistē divi šabloni `{int} {,}` un `( {int} {,} ) *`, tad tokenu virkne `{int:4} {,} {int:6} {,}` tiks akceptēta ar otru šablonu, neskatoties uz to, ka augstākās prioritātes šablona sakritība tika konstatēta agrāk.

### 4.4. Realizācijas pamatojums

Galvenais princips prototipa risinājuma izstrādē bija izveidot to tā, lai jebkurā laika momentā šablonu sakrišanu meklēšanai būtu nepieciešams lineārs laiks un tikai viena pāreja pa tokenu virkni. Šāda pieeja ir izvēlēta ar iedomu, ka makro pievienošana tiks izpildīta tikai vienreiz, bet sakrišanu meklēšana tiks pildīta katrā produkcijā, un, sliktākajā gadījumā, katram tokenam no ieejas plūsmas.

Ielasītās regulārās izteiksmes tiek pārsētas un pārveidotas nedeterminēta galīgā automātā. Tad katrs no nedeterminētiem galīgiem automātiem tiek determinizēts un minimizēts. Tātad katrai regulārai izteiksmei tiek izveidots minimāls determinēts automāts, kurš ir optimizēts gan pēc laika, gan pēc aizņemtās vietas.

Tālāk, lai nodrošinātu visu šablonu pārbaudi, ir nepieciešams apvienot izveidotos automātus. To var izdarīt dažos veidos. Vienkāršākais no tiek būtu glabāt visus galīgos automātus sarakstā. Pieņemsim, ka ir  $n$  šabloni, kurus vajag pārbaudīt. Tad automātu saraksts reprezentē nedeterminētu galīgu automātu ar  $n$   $\varepsilon$ -zariem no sākuma stāvokļa, katrs no kuriem ved pie sākuma stāvokļa vienam no jau izveidotiem determinētiem automātiem.

Cits veids, kā to varētu izpildīt, ir apvienot visus izveidotos šablonu automātus vienā determinētā galīgā automātā. Tieši šis veids tika izvēlēts šī darba ietvaros lai pēc iespējas samazinātu laiku sakrišanu meklēšanai. Kaut arī automātu apvienošana šādā veidā ir laikietilpīga, tā samazina laika kārtu sakritību meklēšanai. Šāda izvēle ir balstīta uz faktu, ka makro daudzums būs samērā neliels, bet gan sakrišanu meklēšanas izsaukums var notikt katras produkcijas ietvaros, t.i. katra tokena apstrādē.

Diemžēl pilnībā lineāra laika sakrišanu meklēšana nav iespējama tādēļ, ka prototips dod iespēju lietot makro ar tokenu vērtībām. Piemēram, eksistē 2 makro, viens no kuriem vēlas saņemt tokenu `{id}`, un otrs `{id:foo}`. Gadījumā, kad sakrišanu meklēšanas procesā parādās tokens `{id:foo}`, automātam nav iespējas izsecināt, kurš no ceļiem novedīs pie garākas sakritības. Tādēļ tas iet pa abiem ceļiem vienlaikus, saglabājot abus stāvokļus.

Automātu minimizēšana ir diezgan darbietilpīga operācija (sk. 4.6.3.), tāpēc tā tiek izpildīta tikai uz atsevišķiem automātiem. Liels automāts nebūs minimāls, jo dažādu minimālu automātu apvienošana negarantē šo faktu, bet tā kā apvienota automāta stāvokļu daudzums var būt ļoti liels, minimizēšanas izpilde var būt neefektīva. Tā kā minimizēšana samazina tikai automāta aizņemto vietu, nevis apstaigāšanas laiku, to var neievertot.

## 4.5. Kāpēc neder jau uzrakstītas bibliotēkas

Šinī darbā aprakstītai apstrādei neder jau izstrādātas regulāro izteiksmju bibliotēkas, kaut arī ir izstrādātas daudzas bibliotēkas, gan parastās, gan ar līdzīgu pieeju, ko apskata šīs darbs (piem. [1], kas lieto automātu teoriju lai pāātrinātu apstrādes laiku). Tās neder tādēļ, ka regulāro izteiksmju dzinēji strādā ar tekstu, nevis ar tokeniem. Tie arī parasti piedāvā daudz vairāk funkcionalitātes, nekā ir nepieciešams, piemēram, meklēšana no virknes sākuma. Citu programētāju rakstītā koda pārstrāde aizņemtu vairāk laika nekā tā izveidošana pilnīgi no jauna.

Jau eksistējošās bibliotēkas nepiedāvā iespēju apvienot visus automātus viena vienīgā. Eksistējošie apvienošanas risinājumi, savukārt, pēc apvienošanas neņem vērā, tieši kāda regulārā izteiksme uz doto brīdi ir akceptējosa. Šī darba ietvaros šīs informācijas saglabāšana ir būtiska, jo no tās ir atkarīgs, kāda no transformācijām ir lietota.

Apvienošanas pieeja parasti ir lietota lielu datu plūsmu apstrādē, piemēram, meklējot infekcijas pazīmes plūsmā. Tad izstrādātās regulārās izteiksmes ir optimizētas un apvienotas vienā automātā. Šādā gadījumā nav nozīmes, kura no izteiksmēm ir akceptēta, jo galvenais ir identi-

ficēt draudu pazīmes.

## 4.6. Lietotie algoritmi

Šī apakšnodaļa apraksta pieejas un algoritmus, kas tika lietoti prototipa realizācijā. Kā jau bija teikts, meklēšanas laika optimizācijai tika izvēlēta pieeja, kur visi regulāro izteiksmju automāti tiek sapludināti kopā.

Visu automātu pārejas pa zariem notiek nevis pa kādu simbolu, bet gan pa tokeniem.

### 4.6.1. Regulāro izteiksmju pārveidošana nedeterminētā galīgā automātā

Regulāro izteiksmju translēšana uz nedeterminētu galīgu automātu ir diezgan vienkārša. Ir nepieciešams tikai pārveidot galvenos kontroles elementus. Tā kā uz doto brīdi prototips atbalsta tikai ierobežotu regulāro izteiksmju sintaksi, to ir vienkārši izdarīt.

Nedeterminēts galīgs automāts (NGA) veselai regulārai izteiksmei ir izveidots to daļējiem automātiem katrai regulārās izteiksmes daļai. Katram operatoram tiek piekārtota attiecīga konstrukcija. Daļējiem automātiem nav akceptējošu stāvokļu, tiem ir pārejas uz nekurieni, kuras vēlāk tiks lietotas lai savienotu automāta daļas. Pilnīga automāta būvēšanas process beigsies ar akceptējošā stāvokļa pievienošanu palikušajām pārejām.

NGA vienam tokenam *token* izskatīsies sekojoši:

**FIXME:** *picture*

NGA divu automātu konkatenācijai  $e_1e_2$  izskatīsies sekojoši:

**FIXME:** *picture*

NGA izvēlei starp diviem automātiem  $e_1|e_2$  izskatīsies sekojoši:

**FIXME:** *picture*

NGA priekš konstrukcijas  $e_1*$  izskatīsies sekojoši:

**FIXME:** *picture*

Tālāk šie automāti tiek apvienoti vienā, piemēram izteiksmei  $\{id\} (\{real\} | \{int\}) * \{;\}$  NGA izskatīsies sekojoši:

**FIXME:** *picture*

[4], [2]

### 4.6.2. Determinizācija

[2]



### 4.6.3. Minimizācija

**FIXME:** *Uzrakstīt!*

### 4.6.4. Apvienošana

**FIXME:** *Uzrakstīt!*

### 4.6.5. Sakrišanu meklēšana

**FIXME:** *Uzrakstīt*

### 4.6.6. Tvērumi

Viena no galvenām šīs sistēmas īpašībām ir iespēja atšķirt programmatūras tvērumus. Ir dažādi veidi, kā var izveidot automātus, kas atšķirtu atsevišķu tvērumu makro. Viens no veidiem varētu būt tāds - katram tvērumam izveidot automātu rindu, kur tvērumam specifiskākie automāti tiks pārbaudīti pirmie. Bet gadījumā, ja ir  $n$  iekļautie tvērumi un nepieciešamais makro ir atrodamas pirmajā automātā, būs jāizpilda vismaz  $n$  meklēšanas, līdz ko pareizais šablons tiks atrasts. Atstājot tikai vienu aktīvu automātu vienā laika brīdī, arī ir dažas iespējas. Varētu visus šablonus likt vienā automātā kopā, un tad pēc izejas no tvēruma attiecīgos šablonus dzēst ārā. Tas nozīmētu, ka katram tvērumam jāatceras makro, kas tika pievienoti, un jāprot dzēst daļu no stāvokļiem ārā no automāta. Bet stāvokļu dzēšana ir laikietilpīga operācija, jo tās izpildīšanai būs nepieciešams apstaigāt visu lielo automātu, dzēšot no tā nevajadzīgos stāvokļus.

Tāpēc tika izvēlēta sekojoša pieeja. Ja prototips darba gaitā sastapās ar tvēruma sākuma simbolu, tas izveido eksistējošā automāta kopiju un ieliek to kaudzē. Tad automātam tiek pievienotas tvēruma makro. Izejot no tvēruma tā specifiskais automāts tiek izmests ārā un darbs tiek turpināts ar pēdējo automātu no kaudzes, kas atbilst iepriekšējam tvērumam. Šādā veidā jebkurā laika brīdī aktīvs ir tikai viens sakrišanu meklēšanas automāts.

## 4.7. Izņēmumi

### 4.7.1. Transformācijas

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

### 4.7.2. 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.

#### 4.7.3. Tokenu mantošana

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 jārūpējas programmētājam rakstot savas makro izteiksmes.

#### 4.7.4. Regulārās izteiksmes daļu grupēšana

Tādēļ, ka aprakstītā pieeja mēģina pēc iespējas minimizēt meklēšanas laiku, tā neatļauj veidot atrasto tokenu grupēšanu, kā tas ir parasti pieņemts regulārās izteiksmēs. Atpakaļnorādes (*backreferences*) uz tokenu grupām nav atļautas.

Tomēr nepieciešamības gadījumā ir apskatīta arī tāda iespēja. Tas varētu tikt izpildīts, determinizējot tikai automāta stāvokļu grupu iekšienē, pēc tam ar  $\epsilon$ -pārejām secīgi savienojot grupu automātu akceptējošus un sākuma stāvokļus. Automāts nebūs pilnīgi determinizēts, tikai grupu ietvaros, bet tad būs pieejamas atrasto tokenu grupas. Tomēr šāda pieeja neatļauj sapludināt dažus automātus vienā, jo šāda procedūra sabojās grupēšanu.

#### 4.7.5. Tokenu vērtības

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

### 4.8. Optimizācijas iespējas

Regulāro izteiksmju optimizēšana netiek izpildīta, jo to ir vērts izpildīt uz regulārām izteiksmēm, kas tiks lietoti daudzas reizes. Darba apskatītā situācijā regulārās izteiksmes tiks lietotas tikai vienas programmas ietvaros un to optimizēšanai nav īpašas jēgas. Tomēr bez reāliem piemēriem nevar izšķirt, vai tas izveidos būtisku paātrinājumu darbā vai nē.

## 5. Rezultāti

### 5.1. 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?*

## 6. Secinājumi

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

# Literatūra

- [1] Re2 regular expression parser. <http://code.google.com/p/re2/>.
- [2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.
- [3] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25(11):35--44, November 1990.
- [4] Russ Cox. Regular expression matching can be simple and fast. 1 2007.
- [5] 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.
- [6] John N. Shutt. Recursive adaptive grammars. Master's thesis, Worcester Polytechnic Institute, August 1993.