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

Termini un apzīmējumi

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

Regulārās izteiksmes

Atpakaļnorādes

Priekšprocesors

Tvērums 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.

Tipu izsecināšana

ε-pārejas

Nedeterminēts galīgs automāts

Determinēts galīgs automāts

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 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. Termināliе 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 netermināliem simboliem, bet b ir neterminālu un terminālu simbolu virkne. Kad kāda likuma kreisē puse parādās apstādāmo simbolu rindā, rinda var tikt pārrakstīta aizvietojot kreisēs puses netermināli ar labo likuma daļu. $\mathtt{A} \underset{G}{\overset{*}{\Rightarrow}} \mathtt{b}$ parāda, ka $\mathtt{A}$ var tikt pārveidots virknē $\mathtt{b}$ lietojot gramatikas G pārrakstīšanas likumus. Šādas secīgu pārveidojumu rinda tiek saukta par atvasinājumu. [**?**]

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 netermināliem 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. [5]

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.[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 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}`. [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 tvērums ($lexicalscope$) un statiskā tipizācija ($statictyping$).

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

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. 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.

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.

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 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 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 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.

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..

```
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

3.4. Transformācijas pieeja

Šī nodaļa parāda sistēmas sadalīšanu uz trim neatkarīgam daļām. Pirmā no tām ir sakrišanu meklēšanas daļa. Tā tokenu virknē atrod makro šablonu satikšanas reizes. Otrā ir transformācijas daļa. Tā pārveido sakrišanas mehānisma atrasto tokenu virkni atbilstoši tam, kas norādīts makro labajā daļā. Trešā ir tipu pārbaudīšanas sistēma, kas statiski pārbauda, vai uzrakstītais makro vispār ir derīgs valodas gramatikas ietvaros. Šīs sadalījums ir tikai konce

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 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 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 transformācijas 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 | 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.

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 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 darboša-

nās 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 **??**..

4. Prototipa realizācija

Šī darba ietvaros tika izstrādāts prototips sakrišanu meklēšanas sistēmai. Šī nodaļa apraksta prototipa īpašības, kā arī pieejas un algoritmus, kas tika lietoti tā realizācijā. Prototips vienkāršības un izstrādes ātruma dēļ tika rakstīts Python valodā, un ir viegli palaižams un atkļūdojams uz jebkura datora ar pieejamu 2.7. Python versiju.

Prototips tika izstrādāts ar iedomu pēc iespējas samazināt sakrišanu meklēšanas laiku, jo transformācijas sistēmas izsaukumi notiks katras produkcijas apstrādē.

Šīs nodaļas organizācija ir sekojoša. Apakšnodaļa 4.1. īsi apraksta prototipam pieejamu sintaksi. Apakšnodaļa 4.2. apskata darbību virkni, ko prototips izpilda darba gaitā. Tālāk apakšnodaļa 4.3. apskata izvēlētās metodes šablonu konfliktu risināšanai. Apakšnodaļā 4.4. ir aprakstīts, kāpēc tika izvēlēta šāda realizācijas pieeja un apakšnodaļa 4.5. apskatīts, kāpēc dotajā gadījumā nav derīgi kādi gatavie risinājumi. Apakšnodaļa 4.6. stāsta par realizācijā lietotiem algoritmiem, bet apakšnodaļa 4.7., savukārt, apraksta problēmas ar kurām saskārās darba autors un izņēmumus, kas pagaidām netiek implementēti prototipā.

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 specificētu tokenu vai pseido-tokenu vērtībām. Piemēram, regulārā izteiksme `{id:foo}` sagaidīs tieši identifikatoru `foo`, bet izteiksme `{id}` sagaidīs jebkuru identifikatoru. Tas ievieš dažādas problēmas, kas tiks aprakstītas zemāk, bet dod lielāku brīvību šablonu sistēmas lietotājam.

4.2. Vispārīgā pieeja

Prototips imitē darbu reālajā vidē, saņemot tokenus no ieejas plūsmas pa vienam no atsevišķas saskarnes. Tokenu plūsmas saskarne imitē leksera darbu. Kamēr prototips nav saņēmis nevienu regulāro izteiksmi, tas ignorē tokenu plūsmas apstrādes izsaukumus. Tiklīdz prototipam atnāk izsaukums apstrādāt tokenu regulāro izteiksmi, tas uzsāk regulārās izteiksmes parsēšanu. Parsēšanas procesā tiek izveidots galīgs automāts, kas akceptē regulārās izteiksmes uzdotās virknes. Katra jauna regulāra izteiksme izveido jaunu automātu.

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 akceptētā makro šablona noteikumiem. Tad sakrišanas meklēšanas sistēmas darbs tiek uzsākts no jauna 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, ja tika atrastas sakrišanas. Kad sistēma tiks integrēta ar reālu kompilatoru, tā strādās paralēli ar parsētāju un tālāk 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, kad divas izteiksmes atpazīst vienu un to pašu virkni vienā tvērumā. Otrais var rasties gadījumā, kad 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 ar identifikatoriem `1` un `2`, bet otrajā tvērumā tiks ieviestas makro ar identifikatoriem `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 neka tie, kas atrodami agrākā tvērumā.

4.3.3. Dažādu virkņu garumu konflikts

Prototips strādā pēc mantkārīga (*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 uz kura bāzējas prototipa izstrāde ir samazināt apstrādes laiku. Tātad prototipa risinājums tika izveidots tā, lai jebkurā laika momentā šablonu sakrišanu meklēšanai būtu nepieciešams lineārs laiks un tikai viena tokenu virknes apstaigāšana. Šāda pieeja ir izvēlēta ar iedomu, ka makro pievienošana tiks izpildīta tikai vienreiz, un to daudzums būs samērā neliels, bet sakrišanu meklēšana tiks pildīta katrā produkcijā, un, sliktākajā gadījumā, katram tokenam no ieejas plūsmas.

Ielasītā regulārā izteiksme tiek pārsēta un pārveidota nedeterminēta galīgā automātā. Tad šablona nedeterminēts galīgs automāts tiek determinēts un minimizēts. Tātad katrai regulārai izteiksmei tiek izveidots minimāls determinēts automāts, kurš ir optimizēts gan pēc apstaigāšanas laika, gan pēc aizņemtās vietas.

Tālāk, lai nodrošinātu visu šablonu pārbaudi vienlaikus un minimizēt meklēšanas laiku, 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 ε-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 šīs 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.

4.5. Kāpēc neder jau uzrakstītas bibliotēkas

Eksistē diezgan daudz jau izstrādātu bibliotēku, kas apstrādā regulārās izteiksmes. Piemēram, darbs [1] arī lieto automātu teoriju lai paātrinātu apstrādes laiku. Diemžēl tās nav lietojamas tādēļ, ka visi eksistējošie regulāro strādā tieši ar tekstu un simboliem, nevis ar tokeniem un to virknēm. Eksistējošo bibliotēku pārtulkošana uz tokenu apstrādi būtu darbietilpīgāka nekā jaunā moduļa izveide. Šādas bibliotēkas parasti arī piedāvā daudz vairāk funkcionalitātes, nekā ir nepieciešams, piemēram dažādi kontroles simboli, tādi ka ^ - meklēt no virknes sākuma. Tā kā sistēma tiks izsaukta patvaļīgas tokenu virknes vietas apstrādei, šāda veida kontrole nav nepieciešama. Citu programmētāju rakstītā koda apārstrāde un vienkāršošana aizņemtu vairāk laika nekā tā izveidošana pilnīgi no jauna.

Pieejamas regulāro izteiksmju bibliotēkas arī nepiedāvā iespēju apvienot visus automātus viena vienīgā. Eksistējošie apvienošanas risinājumi, savukārt, pēc apvienošanas neņem vērā, tieši kāda regulārā izteiksme uz doto brīdi ir akceptējoša. Šī darba ietvaros šīs informācijas saglabāšana ir būtiska, jo no tā ir atkarīgs, kāda no transformācijām ir lietota.

4.6. Lietotie algoritmi

Šī apakšnodaļa apraksta 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 attiecīgu tokenu. Regulāro izteiksmju apstrādes gaitā tokeni `{id:foo}` un `{id}` tiek uzskatīti par dažādiem, kaut arī `{id:foo}` ir apakšgadījums tokenam `{id}`. Šīs fakts tiek iegaumēts tikai sakrišanu meklēšanas gaitā.

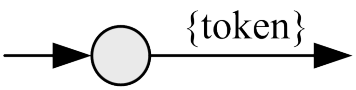
4.6.1. Regulāro izteiksmju pārveidošana nedeterminētā galīgā automātā

FIXME: *Proof that they have the same computational power*

Regulāro izteiksmju translēšana uz nedeterminētu galīgu automātu ir diezgan vienkārša. Lai to paveikt ir nepieciešams pārveidot galvenos regulārās izteiksmes kontroles elementus un automāta gabaliem. Tā kā uz doto brīdi prototips atbalsta tikai ierobežotu regulāro izteiksmju sintaksi, to ir vienkārši izdarīt.

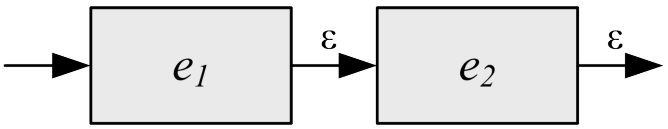
Nedeterminēts galīgs automāts (NGA) veselai regulārai izteiksmei ir izveidots to daļējiem automātiem katrai regulārās izteiksmes daļai. Katram operatoram tiek piekārtota attiecīga konstrukcija. Daļējiem automātiem nav akceptējošu stāvokļu, tiem ir pārejas uz nekurieni, kuras vēlāk tiks lietotas lai savienotu automāta daļas. Pilnīga automāta būvēšanas process beigsies ar akceptējošā stāvokļa pievienošanu palikušajām pārejām. Zemāk tiek parādīti automāti katrai no regulārās izteiksmes iespējamām sastāvdaļām.

Attēlā 5. ir parādīts NGA vienam tokenam $token$.



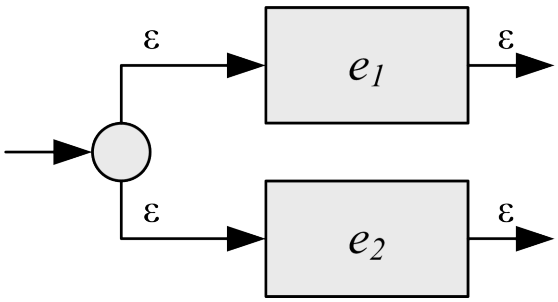
5. att. Automāts vienam tokenam

Attēlā 6. ir parādīts NGA divu automātu konkatenācijai $e_1 e_2$.



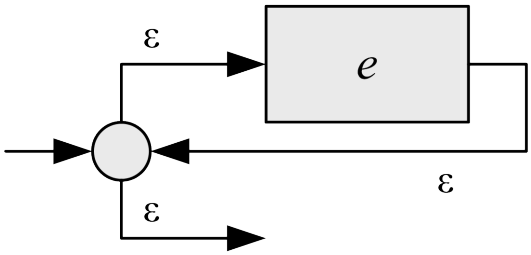
6. att. Automāts divu automātu konkatenācijai

Attēlā 7. ir parādīts NGA izvēlei starp diviem automātiem $e_1|e_2$.



7. att. Automāts izvēlei starp diviem automātiem

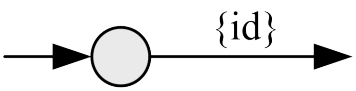
Attēlā 8. ir parādīts NGA priekš konstrukcijas e_1*.



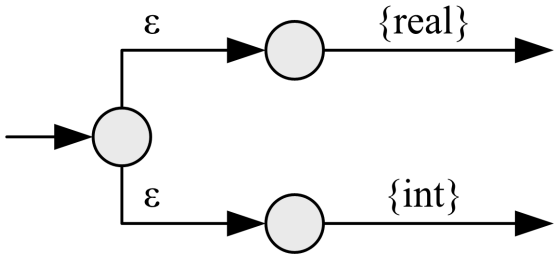
8. att. Automats automātu virknei

Tālāk šie automāti tiek apvienoti vienā, un beigās tiek pievienots akceptējošais stāvoklis.

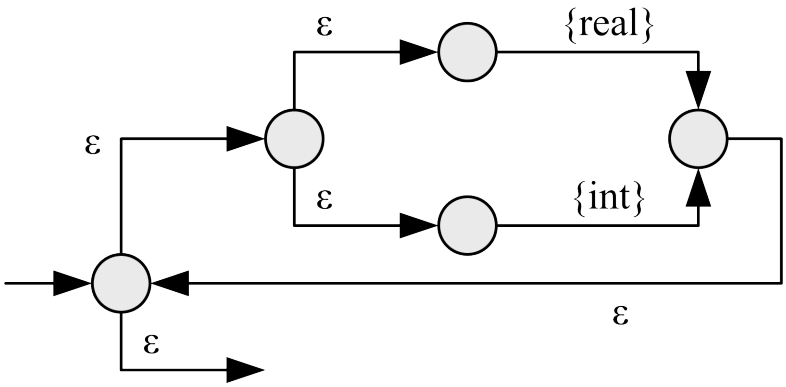
Apskatīsim piemēru - izteiksmi `{id} ({real} | {int}) *`. Sākumā tiek izveidoti NGA izteiksmes daļām. Attēls 9. parāda NGA priekš `{id}`. Attēls 10. parāda NGA priekš daļas `{real} | {int}`. Attēls 11. parāda NGA priekš `({real} | {int}) *`.



9. att. Automats tokenam `{id}`

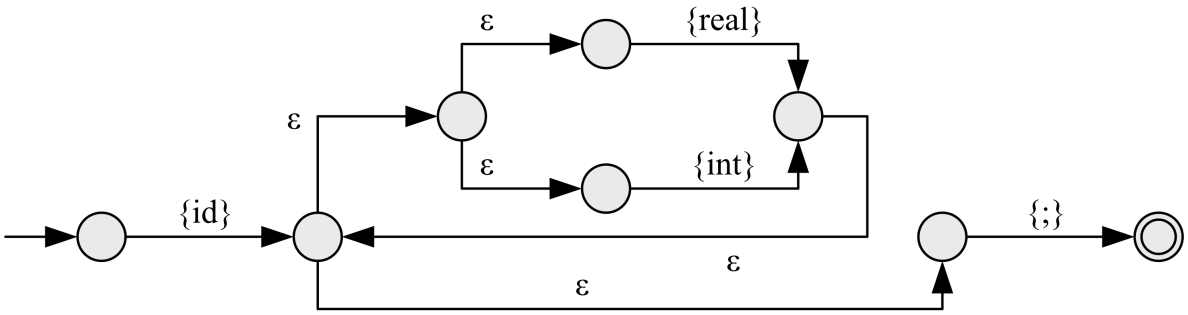


10. att. Automats izteiksmei `{real} | {int}`



11. att. Automats izteiksmei `({real} | {int}) *`

Tad izveidotie automāti var tikt savienoti un beigās tiem tiek pievienots akceptējošais stāvoklis (attēls 12.).



12. att. Automats izteiksmei `{id} ({real} | {int}) *`

Tā tiek izveidots nedeterminēts automāts katrai regulārai izteiksmei. [4], [2]

4.6.2. Determinizācija

Kaut arī daudzām valodām ir vienkāršāk uzbūvēt nedeterminētu galīgu automātu (piemēram, pašām regulārām izteiksmēm tas ir loģiskāk), ir patiess tas, ka katra valoda var tikt aprakstīta gan ar nedeterminētu, gan ar determinētu galīgu automātu. Turklāt, dzīvē sastopamās situācijās DGA parasti satur tik pat daudz stāvokļu, cik ir NGA. Sliktākajā gadījumā, tomēr var gadīties, ka mazākais iespējamais DGA saturēs m^n stāvokļu (m - ieejas alfabēta elementu skaits), kamēr mazākais NGA saturēs n stāvokļus. [1]

Izrādās, ka patiesībā katram NGA eksistē ekvivalents DGA, ko var uzbūvēt ar apakškopu sastādīšanas algoritmu [2]. Vadošā doma šī algoritmā ir tas, ka katrs determinētā galīgā automāta (DGA) stāvoklis ir kādu NGA stāvokļu kopa. Pēc ieejas virknes $a_1, a_2, ..., a_n$ ielasīšanas DGA atrodas stāvoklī, kas atbilst NGA stāvokļu kopai, kuru var sasniegt apstaigājot virkni $a_1, a_2, ..., a_n$.

[1] Pieņemsim, ka automāta valoda sastāv no diviem simboliem - 0, 1. Sliktākais gadījums, kad DGA tiešām saturēs 2^n stāvokļus attiecībā pret n NGA stāvokļiem, var rasties tad, kad, piemēram, automāta valodā n-tais simbols no virknes beigām ir 1. Tad DGA būs jāprot atcerēties pēdējos n simbolus. Tā kā ir divi ieejas alfabēta simboli, automātam ir jāatceras visas to dažādas 2^n kombinācijas.

[2] Pierādījumu tam, ka uzbūvētais DGA tik tiešām akceptē to pašu valodu, ko NGA, sk. [5], teorēma 2.11.

Algoritms 1: NGA transformēšana uz DGA

Ieeja: NGA N.

Izeja: DGA D, kas ir ekvivalents N.

Algoritms: Sākumā algoritms konstruē pāreju tabulu priekš D. Katrs D stāvoklis ir N stāvokļu kopa, tātad tabula tiek konstruēta tā, lai D simulētu vienlaikus visas pārejas, ko var izpildīt N, saņemot kādu ieejas virkni. Lai automāts kļūtu determinēts, ir nepieciešams atbrīvoties no iespējas atrasties dažos stāvokļos vienlaikus. Tātad vajag atbrīvoties no ε-pārejām, un no daudzkārtīgām pārejām no viena stāvokļa pa vienu ieejas simbolu.

Tabulā 1. var redzēt divas funkcijas, kas ir nepieciešamas NGA apstrādes izpildei. Šīs funkcijas no NGA stāvokļiem un pārejām veido jaunas stāvokļu kopas, kuras veidos DGA stāvokļus.

1. tabula. NGA apstaigāšanas funkcijas

Funkcija	Apraksts
$\varepsilon - closure(T)$	NGA stāvokļu kopa, kas ir sasniedzama lietojot tikai ε-pārejas no visiem stāvokļiem no kopas T.
$move(T, a)$	NGA stāvokļu kopa, kas ir sasniedzama lietojot pārejas pa simbolu a no visiem stāvokļiem no kopas T.

Ir nepieciešams apstrādāt visas tādas N stāvokļu kopas, kuras ir sasniedzamas, N saņemot kaut kādu ieejas virkni. Indukcijas bāzes pieņēmums ir tas, ka pirms darbības uzsākšanas N var atrasties jebkurā no stāvokļiem, kurus var sasniegt pārejot pa ε bultiņām no N sākuma stāvokļa. Ja s_0 ir N sākuma stāvoklis, D sākuma stāvoklis būs $\varepsilon - closure(set(s_0))$. Indukcijai pieņemam, ka N var atrasties T stāvokļu kopā pēc virknes x ielasīšanas. Tad, ja N ielasīs nākamo simbolu a, tad N var pārvietoties jebkura no stāvokļiem $move(T, a)$. Taču pēc a ielasīšanas var notikt vēl dažas ε-pārejas, tāpēc pēc virknes xa ielasīšanas N var atrasties jebkurā no stāvokļiem $\varepsilon - closure(move(T, a))$. Attēls 13. parāda pseido-kodu algoritmam, kā šādā veidā var tikt uzkonstruēti visi DGA stāvokļi un tā pāreju tabula.

Automāta D sākuma stāvoklis ir $\varepsilon - closure(set(s_0))$, bet D akceptējošie stāvokļi ir visas tās NGA stāvokļu kopas, kas satur vismaz vienu akceptējošu stāvokli. $Dstates$ ir jauna automāta D stāvokļu saraksts un $Dtran$ ir stāvokļu pāreju tabula.

Sarežģītība: Sarežģītības novērtējums šīm algoritmam ir diezgan nepatīkams. Sliktākajā gadījumā tas būs $O(m^(n + 1))$, kur n ir NGA stāvokļu daudzums un m ir ieejas alfabēta simbolu skaits. Algoritms var uzģenerēt līdz m^n DGA stāvokļiem, katram no kuram ir m pārejas. Taču parasti tas tā nenotiek un DGA stāvokļu skaits ir līdzīgs NGA stāvokļu skaitam, un algoritma sarežģītība ir $O(n * m)$. [2, 5]

initially, $\varepsilon - closure(set(s_0))$ is the only state in $Dstates$, and is unmarked
while there is an unmarked state S in $Dstates$ **do**
    mark S
    for each available oath t from S **do**
        $U = \varepsilon - closure(move(S, a))$
        if U is not in $Dstates$ **then**
            add U as an unmarked state to $Dstates$
        end if
        $Dtran[S, a] = U$;
    end for
end while

13. att. Automāta determinizēšanas algoritms

4.6.3. Minimizēšana

Izveidotais determinēts galīgs automāts var būt neoptimāls pēc stāvokļu skaita. Bet no šī skaitļa ir atkarīgs tālāko soļu izpildes ātrums. Tātad ir nepieciešams izveidot automātu, kas atpazīs to pašu valodu un saturēs minimālu iespējamu stāvokļi skaitu.

Var pierādīt, ka katram automātam eksistē ekvivalents minimāls automāts [1]. Vēl vairāk, ja eksistē 2 dažādi automāti ar vienādu stāvokļu daudzumu, kas atpazīst vienu un to pašu valodu, tad tie ir vienādi līdz stāvokļu nosaukumiem [2].

Tālāk teiksim, ka virkne x atšķir stāvokļu s no stāvokļa t tad, kad tikai viens stāvoklis, ko var sasniegt no t un s pa x ir akceptējošs. Tātad divi stāvokļi ir atšķirami tad, kad eksistē tāda virkne, kas viņus atšķir. Jebkurš akceptējošs stāvoklis ir atšķirams no jebkura neakceptējoša stāvokļa ar tukšu virkni (stāvoklis nevar būt akceptējošs un neakceptējoss vienlaikus). Divi neatšķirami stāvokļi ir ekvivalenti. [3]

Algoritms 2: DGA minimizēšana

Ieeja: DGA D.

Izeja: Jauns DGA D', kas ir minimāls un ekvivalents D.

Algoritms: Minimizēšanas algoritma vadošā doma ir sadalīt automātu neatšķiramos stāvokļu grupās. Tas izveido ekvivalentu stāvokļu grupas, kas tālāk var tikt apvienotas vienā, izveidojot minimāla automāta stāvokļus.

Minimizēšanas gaitā automāta stāvokļi tiek sadalīti grupās, ko uz doto brīdi algoritms nevar atšķirt. Jebkuri divi stāvokļi no dažādām grupām ir atšķirami. Katrā nākamajā algoritma iterācijā eksistējošās grupas tiek sadalītas mazākajās grupās, gadījumā, ja kādā grupā parādās atšķirami stāvokļi. Algoritms apstājas līdz ko neviena grupa nevar tikt sadalīta sīkāk.

[1] Šī fakta pierādījumu sk. ???

[2] Tā kā stāvokļu nosaukumi neietekmē automāta darbību, divi automāti tiek saukti par vienādiem līdz pat stāvokļu nosaukumiem, ja viens no tiem var tikt pārveidots otrajā vienkārši pārsaucot to stāvokļus.

[3] [5] nodaļa 4.4

Pirms algoritms uzsāk darbu, stāvokļi tiek sadalīti divās grupās - akceptējošie stāvokļi un neakceptējošie stāvokļi. Šo grupu stāvokļi ir atšķirami ar tukšu virkni. Tālāk tiek pa vienai apstrādātas grupas no pašreizējā sadalījums. Katrai grupai tiek pārbaudīts, vai tās stāvokļi var tikt atšķirti ar kādu ieejas simbolu - vai kāds no ieejas simboliem noved uz divām vai vairākām dažādam stāvokļu grupām. Ja tādi simboli eksistē, tiek izveidotas jaunas grupas, tādas, ka divi stāvokļi atrodas vienā grupā tad un tikai tad, ja tie aiziet uz vienādām grupām pa vienādiem simboliem. Process ir atkārtots visām pašreizējā sadalījuma grupām, tad atkal jaunam sadalījumam, kamēr neviena no grupām vairs nevar tikt sadalīta.

Attēls 14. parāda minimizēšanas algoritmu pseido-kodā.

initially, partitioning Π contains two groups, F and $S - F$, the accepting and nonaccepting states of D, Π_{new} is empty
while Π_{new} is not equal to Π **do**
$\Pi_{new} = \Pi$
for each group G of Π **do**
partition G into subgroups such that two states s and t are in the same subgroup if and only if for all input symbols a, states s and t have transitions on a to states in the same group of Π
replace G in Π_{new} by the set of all subgroups formed
end for
end while
$\Pi_{final} = \Pi$

14. att. Automāta minimizēšanas algoritms

Tālāk paliek apstrādāt jaunizveidotās stāvokļu grupas izveidojot jaunu determinētu automātu. Lai to izdarītu, no katras sadalījuma Π_{final} grupas tiek izvēlēts grupas pārstāvis. Grupu pārstāvji izveidos jaunus stāvokļus automātam D'. Pārējās komponentes minimālam automātam D' tiks iveidotas sekojoši:

1. Automāta D' sākuma stāvoklis ir pārstāvis tai grupai, kura satur automāta D sākuma stāvokli.

2. Automāta D' akceptējošie stāvokļi ir pārstāvji tām grupām, kuras satur automāta D akceptējošos stāvokļus. Katra no grupām satur vai nu tikai akceptējošus, vai nu tikai neakceptējošus stāvokļus, jo algoritma darba gaitā jaunās grupas tika izveidotas tikai sadalot jau eksistējošas grupas, bet sākuma sadalījums atdalīja šīs stāvokļu klases.

3. Pieņemsim, ka s ir kādas Π_{final} grupas G pārstāvis, un automāts D no stāvokļa s pa ieejas simbolu a pāriet uz stāvokli t. Pieņemsim, ka r ir grupas H pārstāvis, H satur t. Tad automātā D' ir pāreja no stāvokļa s uz stāvokli r pa ieejas simbolu a.

Sarežģītība: Šī algoritma sarežģītība ir $O(n^2 * l)$, kur n ir sākotnējā automāta stāvokļu daudzums un l maksimāls pāreju daudzums no kāda no stāvokļiem [1].

[1] Patiesībā l būtu jābūt ieejas alfabēta elementu skaitam. Bet šī darba ietvaros ir atļauts veidot regulārās izteiksmes ar pārejām pa tokeniem ar specificētu vērtību, piemēram `{int:5}`. Tātad īstenībā ieejas alfabēts ir bezgalīgs.

4.6.4. Apvienošana

Kā jau bija teikts agrāk, lai samazinātu sakrišanu meklēšanas laiku, tika izvēlēts apvienot visus meklēšanas automātus vienā. Tā kā makro atnāk pa vienam dažādās vietās programmas kodā un var sākt uzreiz tikt lietotas, nav iespējams gaidīt kamēr sakrāsies vairāki automāti apvienošanai. Tikko parādās divi automāti tie tūlīt pat tiek apvienoti vienā sistēmā. Tālāk, kad parādās citi makro, to automāti tiek pievienoti jau eksistējošam.

Algoritms pēc savas būtības ir ļoti līdzīgs determinēšanas algoritmam. Vienīgais uzņēmums ir tas, ka nevienā no automātiem neeksistē ε-pārejas. Tātad tas, no kā vajag atbrīvoties, ir pārejas pa vienu un to pašu simbolu uz diviem dažādiem stāvokļiem. Tas tiek darīts apvienojot divus stāvokļus no dažādiem automātiem.

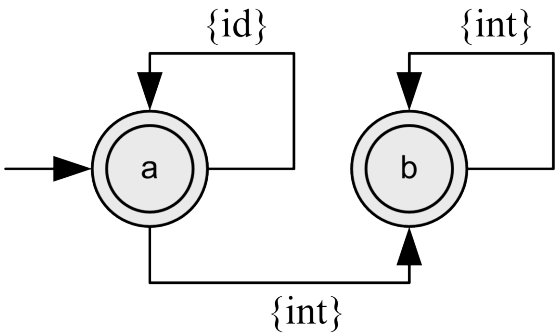
Algoritms 3: Divu DGA apvienošana

Ieeja: DGA D_1 un D_2.

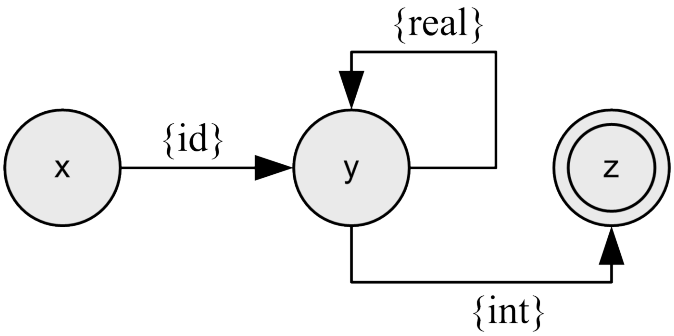
Izeja: Jauns DGA D', kas apvieno D_1 un D_2.

Algoritms: Algoritms sāk darbu apvienojot D_1 un D_2 sākuma stāvokļus. Šo stāvokļu kombinācija veido automāta D' sākuma stāvokli. Tālāk algoritms apskata visas iespējamas pārejas no katra no kombinētiem stāvokļiem un apvieno to rezultātus jaunajos stāvokļos.

Apskatīsim piemēru automātu apvienošanai. Attēls 15. parāda determinētu minimālu automātu priekš regulārās izteiksmes `{id}* {int}*`. Tas satur divus stāvokļus, `a` un `b`, kuri abi ir akceptējoši. Attēls 16., savukārt, parāda DGA priekš izteiksmes `{id} {real}* {int}`.

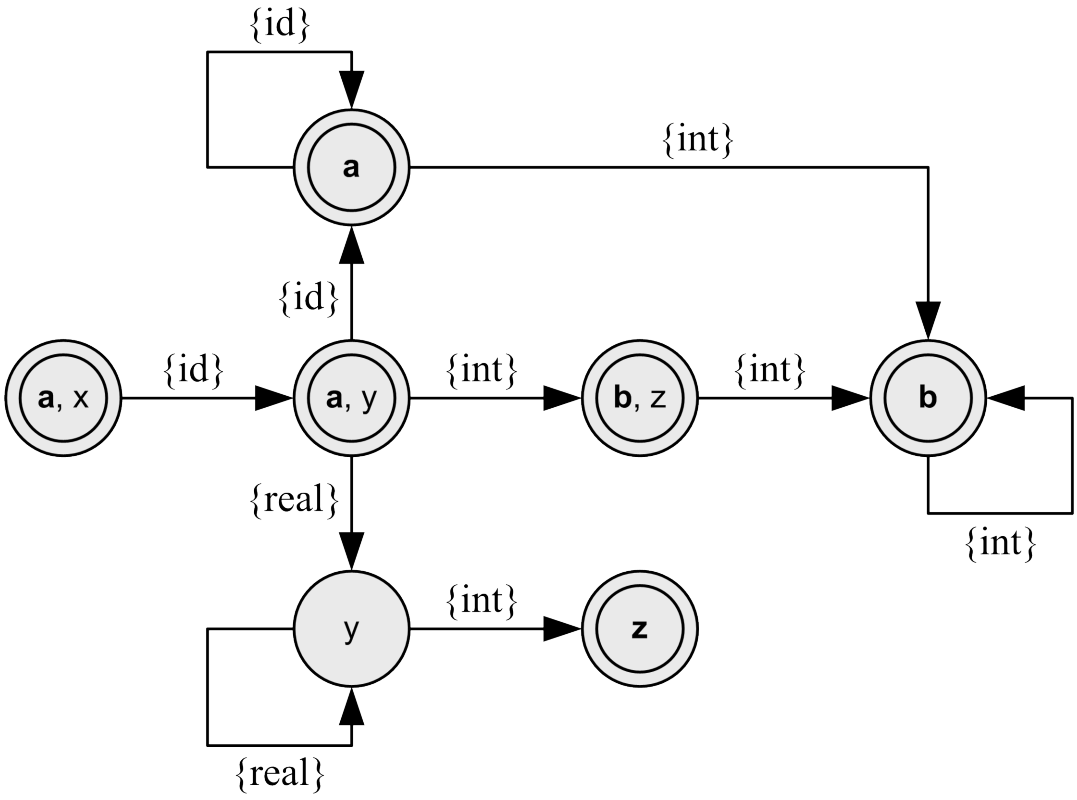


15. att. Automats izteiksmei `{id} ({real} | {int}) *`



16. att. Automats izteiksmei `{id} ({real} | {int}) *`

To apvienotais automāts ir parādīts attēlā 17..



17. att. Automats izteiksmei `{id} ({real} | {int}) *`

Attēls 18. parāda apvienošanas algoritmu pseido-kodā. s_0 ir jaunā automāta D' sākuma stāvoklis, r_0 ir D_1 sākuma stāvoklis un t_0 ir D_2 sākuma stāvoklis. $Dstates$ ir automāta D' stāvokļu saraksts, $Dtran$ ir D' pāreju tabula. Funkcija $move(S, t)$ atgriež tos stāvokļus, uz kuriem var nokļūt no S pa tokenu t. Tā kā parasti S sastāv no diviem stāvokļiem r_i un t_i, tā atgriež pārejas rezultātu no katra no tiem. Rezultāts arī var but tikai viens stāvoklis, gadījumā, ja no kāda r_i un t_i neeksistē pāreja pa doto tokenu.

Sarežģītība: Šī algoritma sarežģītība ir $O(n * m * l)$, kur n ir pirmā automāta stāvokļu daudzums, m ir otrā automāta stāvokļu daudzums, un l maksimāls pāreju daudzums no katra no stāvokļiem.

initially, $s_0 = (r_0, t_0)$ is the only state in $Dstates$, and is unmarked
while there is an unmarked state S in $Dstates$ **do**
    mark S
    for each available move t from S **do** ▷ S is a combination of some states r_i and t_i of D_1 and D_2, although it might be just a single state from one of the automata.
        $U = move(S, t)$
        if U is not in $Dstates$ **then**
            add U as an unmarked state to $Dstates$
        end if
        $Dtran[S, t] = U$;
    end for
end while

18. att. Divu automātu apvienošanas algoritms

4.6.5. Sakrišanu meklēšana

Sakrišanu meklēšana NGA sliktākajā gadījumā būs ar sarežģītību $O(n^2 * l)$, kur n ir NGA stāvokļu skaits un l - pārbaudāmās virknes elementu skaits. Tas var notikt, kad visi NGA stāvokļi ir aktīvi vienā laika brīdī, un no katra no tiem eksistē n pārejas pa ieejas simbolu uz visiem automāta stāvokļiem.

Tīrā DGA gadījumā sakrišanu pārbaude katram ieejas elementam ir $O(l)$, kur l ir pārbaudāmās virknes garums.

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 gaida 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.

Kaut arī tas ievieš nedeterminētību, tā var parādīties tikai augstāk minētā gadījumā un izveidot ne vairāk ka 2 ceļus vienlaikus. Tātad kaut arī nedeterminētība pastāv, tai ir ļoti maza iespējamība un maza ietekme uz sakrišanu meklēšanas laiku.

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 atrodams 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

Izstrādātais prototips nenodarbojas ar tokenu virkņu transformācijām, jo tā nav sakrišanu meklēšanas mehānisma uzdevums. Tālākajā sistēmas izstrādes gaitā notiks integrācija ar transformāciju sistēmu vai arī abu mehānismu sapludināšana vienā.

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 ieviesta vēlāk, kad tiks uzsākta integrācija un sadarbība ar reālu parsētāju. Tā varētu tikt implementēta līdzīgi tam, kā tiek realizēti konteksti - pa vienam sapludinātam automātam priekš katra produkcijas tipa.

4.7.3. Tokenu klašu mantošana

Sistēmai nav nekādas informācijas 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 sistēmas lietotājam 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. Tokenu grupēšana un atpakaļnorādes (*backreferences*) uz tokenu grupām nav atļautas.

Tomēr ja parādīsies nepieciešamība, ir apskatīta arī pieeja, kas dos šādu iespēju. Tas varētu tikt izpildīts, determinējot tikai automāta stāvokļus grupu iekšienē, pēc tam ar ε-pārejām secīgi savienojot grupu automātu akceptējošus un sākuma stāvokļus. Automāts būs determinēts tikai grupu ietvaros, bet tad būs pieejamas atrasto tokenu grupas. Tomēr šāda pieeja neatļaus sapludināt dažus automātus vienā, jo sapludināšanas procedūra sabojās grupēšanu.

4.7.5. Sapludinātā automāta minimizācija

Automātu minimizēšana ir diezgan darbietilpīga operācija, tāpēc tā tiek izpildīta tikai uz atsevišķiem automātiem. Apvienotais visu šablonu automāts var nebūt minimāls, jo dažādu mi-

nimālu 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 šajā gadījumā var izlaist. Sapludinātā automāta minimizēšanu apgrūtina arī tas fakts, ka to stāvokļus vajadzēs atšķirt arī pēc tā fakta, kāds no šabloniem ir akceptēts, nevis tikai pēc tā, vai stāvoklis ir akceptējošs.

4.8. Optimizācijas iespējas

Regulāro izteiksmju optimizēšana uz doto brīdi 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 šādā konkrētā gadījumā vai nē. Dažas regulāro izteiksmju pārrakstīšanas pieejas ir aprakstītas [6].

Dažreiz divu automātu sapludināšana var izraisīt pārāk lielu stāvokļu daudzumu rašanos. Ja reālajā situācijā tas ietekmēs apstrādes laiku, vai parādīsies nepieciešamība ierobežot automāta aizņemto laiku, var apskatīt iespēju glabāt dažus automātus viena vietā. [6]

Minimizēšanas algoritmu var aizvietot ar citu, ātrāku algoritmu, piemēram, Hopkrofta minimizēšanas algoritmu, kura izpildes laiks ir $O(n \log n\, l)$, kur n ir automāta stāvokļu daudzums un l ir ieejas alfabēta elementu daudzums.[3]

5. Līdzīgu darbu apskats

FIXME: *related work*

6. Rezultāti

Šī darba ietvaros tika izstrādāts prototips sakrišanu meklēšanas sistēmai. Šīs nodaļas apakšnodaļa 6.1. apskata testēšanas stratēģijas, kas tika lietotas lai nodrošinātu to, ka prototips strādā pareizi. Apakšnodaļa 6.2. apraksta integrēšanas iespējas ar valodas Eq kompilatoru.

6.1. Prototipa testēšana

Prototips tika testēts visā izstrādes laikā. Zemāk tiks aprakstīti automātiski palaižamie testi.

6.1.1. Stresa testēšana

Prototips izstrādes laikā tika testēts ar lieliem automātiski ģenerēto datu apjomiem. Tika ģenerētas patvaļīgas (bet korektas) regulāras izteiksmes ar iekavu un $*$ un $|$ simbolu palīdzību. Katrai regulārai izteiksmei tika izveidota arī simbolu virkne, ko šai izteiksmei jāprot atpazīt. Tad uz vienas un tās pašas regulārās izteiksmes un simbolu virknes tika palaists gan prototips, kas tolaik apstrādāja simbolus, gan Python iebūvētais regulāro izteiksmju apstrādes mehānisms. Vienā piegājienā tika ģenerēti 500 šādi testi. Prototipa beigu izstrādes posmā visi šādi testi tika veiksmīgi izpildīti.

Diemžēl pagaidām šī pieeja netiek implementēta ar tokenu regulārām izteiksmēm, jo nav iespējams pārbaudīt sistēmas darba ekvivalenci ar kādu citu sistēmu. Tāpēc tika veikta intensīva sistēmas testēšana, lai pārbaudītu pēc iespējas vairāk reālajā darbā iespējamo situāciju. Tomēr šādas stresa pārbaudes parādīja ka pats regulāro izteiksmju apstrādes mehānisms strādā korekti.

6.1.2. Sistēmas testēšana

Prototipam tika izveidoti apmēram 20 testi, kas pārbauda to darbību iespējamās situācijās. Tabula 2. parāda konceptuālu testu sadalījumu pa grupām. Dažas grupas pārklājas, jo, piemēram, tvērumu pārbaudošie testi pārbauda arī korektas regulāro izteiksmju prioritātes.

2. tabula. Prototipa testu sadalījums pa grupām

Testa nosaukums	**Testa apraksts**	**Kas tiek pārbaudīts**
Testi prioritāšu pārbaudei		
Testi ar dažiem šabloniem vienā tvērumā	Testu gaitā tiek izveidotas dažādas regulāras izteiksmes, kuru aprakstītās valodas pārklājas. Tās tiek ievietotas vienā tvērumā pēc kārtas. Tad tiek pārbaudīts, ka tokenu saraksts tiek akceptēts ar pareizu šablonu attiecībā pret to prioritātēm.	Vai tiek korekti apstrādātas šablonu prioritātes viena tvēruma ietvaros.

2. tabula. Prototipa testu sadalījums pa grupām

Testa nosaukums	**Testa apraksts**	**Kas tiek pārbaudīts**
Testi ar dažiem šabloniem vienā tvērumā, kur kāds no šabloniem akceptē garāku virkni	Testu gaitā tiek izveidotas dažādas regulāras izteiksmes, kuru aprakstītās valodas pārklājas. Tās tiek ievietotas vienā tvērumā pēc kārtas. Tad tiek pārbaudīts, ka tiek akceptēta garākā iespējamā tokenu virkne.	Vai tiek korekti apstrādātas šablonu prioritātes viena tvēruma ietvaros.
Testi tokenu vērtībām		
Testi ar tokenu vērtībām	Testu gaitā tiek izveidotas dažādas regulārās izteiksmes ar tokenu vērtībām un bez tām. Tiem tiek padotas dažādas tokenu virknes.	Vai tiek korekti apstrādātas šablonu prioritātes un vērtību sakrišanas.
Testi ar vairākiem pieejamiem stāvokļiem vienlaikus	Testu gaitā tiek izveidotas dažādas regulārās izteiksmes ar tokenu vērtībām. Tiem tiek padotas tokenu virknes ar šādām pašām vērtībām, lai izveidotu situācijas, kad ir pieejami daži stāvokļi vienlaikus.	Vai tiek korekti apstrādātas situācijas, kad parādās nedeterminētība.
Testi tvērumu pārbaudei		
Testi ar tvērumu iekļaušanas dziļumu 1	Testu gaitā tiek izveidotas dažas regulāras izteiksmes, kuru aprakstītās valodas pārklājas. Tās tiek ievietotas nultajā un pirmajā tvērumā. Tad tiek pārbaudīti tokenu saraksti.	Vai tiek korekti apstrādāta tvēruma parādīšanās. Vai tiek korekti apstrādātas šablonu prioritātes starp tvērumiem.
Testi ar tvērumu iekļaušanas dziļumu >1	Testu gaitā tiek izveidotas dažas regulāras izteiksmes, kuru aprakstītās valodas pārklājas. Tās tiek ievietotas dažādos tvērumos ar dziļumu kas ir lielāks par vienu. Tad tiek pārbaudīti tokenu saraksti.	Vai tiek korekti apstrādāti dažādi tvērumu dziļumi. Vai tiek korekti apstrādātas prioritātes starp tvērumiem.
Testi ar izeju no tvēruma	Testu gaitā tiek izveidotas dažas regulāras izteiksmes, kas tiek ieliktas nultajā un pirmajā tvērumā. Tiek pārbaudīts, ka pirmā tvēruma šablons atpazīst tokenu virkni. Tālāk pirmais tvērums tiek pamests un tiek pārbaudīts, ka tā šablons vairs nav aktīvs.	Vai pēc izejas no tvēruma attiecīgie šabloni ir noteikti izdzēsti.

2. tabula. Prototipa testu sadalījums pa grupām

Testa nosaukums	**Testa apraksts**	**Kas tiek pārbaudīts**
Testi ar izeju no tvēruma un nākamā tvēruma izveidi	Testu gaitā tiek izveidots 1. līmeņa tvērums ar regulārām izteiksmēm. Tiek pārbaudīts, ka pirmā tvēruma šabloni atpazīst tokenu virknes. Tad šīs tvērums tiek pamests un tiek izveidots jauns pirmā līmeņa tvērums. Tiek pārbaudīts, ka vecā tvēruma šabloni ir izmesti, un ka jaunā tvēruma šabloni tiek atpazīti.	Vai pēc izejas no tvēruma attiecīgie šabloni ir izdzēsti un pēc ieejas jaunajā tvērumā tiek akceptēti pareizi šabloni.
Testi bez šablonu sakritībām		
Testi bez neviena šablona	Testu gaitā tiek izveidota sistēma bez neviena šablona. Tiek pārbaudīts, ka neviena sakritība netiek atrasta.	Vai sistēma korekti apstrādā situāciju, kad nav neviena šablona.
Testi ar šabloniem un datiem kas nesakrīt	Testu gaitā tiek izveidota sistēma ar dažiem šabloniem. Tad tiek padotas tokenu virknes kuras neder nevienam no eksistējošiem šabloniem.	Vai sistēma korekti apstrādā situāciju, kad neviena sakrišana nav atrasta.

6.2. Prototipa integrēšana Eq

Prototips pagaidām netiek integrēts Eq valodas kompilatorā, bet tas tiek plānots tuvākajā nākotnē. Tā kā prototips tika izstrādāts bāzējoties uz Eq parsētāja īpašībām, to būs viegli integrēt eksistējošā kodā. Tā darbs ir gandrīz neatkarīgs no parsētāja darba un neietekmēs jau eksistējošo programmu darbību.

Prototips piedāvā saskarni lai uzsākt jauna tvēruma apstrādi (funkcija `enter_context()`), lai pamestu tvērumu (funkcija `leave_context()`), lai pievienotu makro (`add_match(regexp)`) un lai apstaigātu tokenu virkni meklējot sakrišanas (`match_stream(stream)`). Integrēšanai prototipu būs jāpapildina ar iespēju padot produkcijas tipu tokenu virkņu apstrādes funkcijām. Prototipa tokenu saņemšanas funkcijas būs jāpārslēdz uz parsētāja piedāvāto saskarni tokenu dabūšanai.

Prototipam ir nepieciešama vienkārša saskarne no eksistējošā parsētāja. Parsētājam jādot pieeju pie tokenu virknes lasīšanas, kā arī jāprot aizvietot atrastās tokenu virknes ar citām virknēm, iespējams, ar citu garumu. Parsētājam arī jāprot pārstartēt tokenu virknes lasīšanu no aizvietotas virknes sākuma. Uz doto brīdi visas šīs iespējas ir implementētas Eq kompilatorā.

Apvienojot parsētāja un sakritību meklēšanas prototipu būs nepieciešams ievietot prototipa funkciju izsaukumus katras produkcijas apstrādes sākumā. Gadījumos, kas parsētājs sastapās ar tokenu, kas identificē makro sākšanos, būs nepieciešams izsaukt regulārās izteiksmes parsēšanas funkciju. Savukārt, kad tiek apstrādātas citas produkcijas, būs nepieciešams izsaukt sakrišanu meklēšanu. Abu funkciju izsaukumos būs nepieciešams padot arī produkcijas tipu, lai

prototips varētu atšķirt, kādu no automātiem papildināt vai lietot sakrišanu atrašanai. Prototipa funkcijas būs jāizsauc ari tvērumu pārslēgšanu brīdī, lai tas varētu implementēt tvērumu makro prioritāšu sadalīšanu.

7. 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] Jean Berstel, Luc Boasson, Olivier Carton, and Isabelle Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.

[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] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, ANCS '06, pages 93--102, New York, NY, USA, 2006. ACM.