Latvijas Universitāte
Datorikas fakultāte

Ar regulārām izteiksmēm paplašinātu gramatiku dinamiska parsēšana

Bakalaura darbs

Autors
Jūlija Pečerska

Vadītājs
Guntis Arnicāns
profesors Dr. dat.

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 palielināta līdz vērtībai 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

Nedeterminēts galīgs automāts (nondeterministic finite automaton)

ε-pārejas (ε-transitions) Nedeterminētā galīgā automātā pārejas

Determinēts galīgs automāts (deterministic finite automaton)

Parsēšana (parsing)

Daļiņa (token) Pirmajā kompilēšanas stadijā leksiskais analizators sadala tekstu leksēmās (nozīmīgās simbolu secībās) un katrai leksēmai izveido speciālu objektu, kas tiek saukts par daļiņu (angl. *token*). Katrai daļiņai ir glabāts tips, ko lieto parsētājs lai izveidotu programmas struktūru. Ja ir nepieciešams, tiek glabāta arī daļiņas vērtība, parasti tā ir norāde uz

elementu simbolu tabulā, kurā glabājas informācija par daļiņu - tips, nosaukums. Simbolu tabula ir nepieciešama tālākā kompilatora darbā lai paveiktu semantisko analīzi un koda ģenerāciju. Šajā darbā vienkāršības dēļ tiks uzskatīts, ka daļiņas vērtības ailītē glabāsies leksēma, ko nolasīja analizators. Tālāk daļiņas tiks apzīmētas šādā veidā:

```
{token-type : token-value}
```

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

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

Tātad izejas kods `sum = item1 + 5` pēc leksiskās analīzes tiks pārveidots daļiņu plūsmā `{id:sum}, {=}, {id:item1}, {+}, {int:5}`. Nolasīto daļiņu virkne tiek padota parsētājam tālākai apstrādei un abstraktā sintaktiskā koka izveidei. [2]

1. Ievads

Mūsdienu programmēšanas valodas ir spēcīgi rīki, kas var tikt pielietojami dažādu uzdevumu risināšanai. Abstrakti programmēšanas valodas var sadalīt divās grupās - universālas un domēn-specifiskas valodas. Universālas valodas ir pielietojamas visās sfērās, bet domēn-specifiskas gan tiek veidotas kādas konkrētas uzdevumu klases risināšanai. Kaut arī universālas valodas var tikt lietotas šīs uzdevumu klases risināšanai, tās ne vienmēr ir tikpat izteiksmīgas.

Taču parasti tieši universālas programmēšanas valodas tiek lietotas lai risinātu specifiskus uzdevumus, jo ne visiem uzdevumiem eksistē domēn-specifiskas valodas. Domēn-specifiskas valodas ir rīks, ko ir vērts izstrādāt veselai klasei specializētu uzdevumu. Bet to izveide ir liels darbs, ko nav vērts pildīt, lai atrisināt vienu konkrētu uzdevumu.

Gandrīz jebkādas universālas programmēšanas valodas virzošs spēks ir funkcijas un to kompozīcijas. Katra valoda iekļauj sevī kaut kādu bāzes funkcionalitāti, funkciju ietvaru, kuru kompozīcijas un virknes veido programmas. Taču specifisku uzdevumu gadījumā funkciju kompozīcijas nepietiekami apraksta izpildāmo darbību. Dažas darbības ir ērtāk un saprotamāk (tas var būt pieņemts ārpus programmēšanas, reālās dzīves situācijās) attēlot ar citu sintaktisku formu, nevis ar funkciju. Šādas darbības piemērs varētu būt faktoriāla pieraksts - `n!`, kas ir vispārpieņemts apzīmējums.

Viena no šādām sintaktiskām formām ir operatori, tomēr operatoru skaits un pārslogošanas iespējas valodās ir diezgan ierobežotas. To pārslogošana, piemēram, C vidē, neļaus attēlot faktoriālu ar `!` zīmi, jo pārslogojot saglabājas tā prefiksā forma. Tātad operatoru pārslogošana arī nedod brīvību veidot konstrukcijas, kas varētu paaugstināt valodas izteiksmību. Bet citu rīku, kas varētu to piedāvāt, arī nav, jo universālas programmēšanas valodas bieži iekļauj ļoti ierobežotas sintakses mainīšanas iespējas.

Uz doto brīdi vairākumam mūsdienīgo programmēšanas valodu eksistē standarti. Pateicoties šim faktam, ir iespējams izstrādāt dažādus kompilatorus vienai un tai pašai valodai. Bet sintakses izmaiņu ieviešanas gadījumā ir jāmaina standarts, tātad ir jākoriģē arī visi valodas kompilatori. Dažreiz šīs izmaiņas ir tik nopietnas, ka tiešām prasa ievērojamas parsētāja modifikācijas, lai tiktu atbalstītas, bet dažreiz tās ir tikai sintaktiskas, piem. sintaktiskā cukura[1] ieviešana.

Gadījumos, kad ir nepieciešams paplašināt valodu tā, lai tā būtu piemērotāka konkrētam uzdevumam, sintakses izmaiņas var būt nebūtiskas. Tomēr vairākums valodu neļaus šādas izmaiņas ieviest bez valodas gramatikas, un, secīgi, valodas standarta modifikācijas.

Šīs darbs izpēta jautājumu, kā ir iespējams iznest valodas sintakses izmaiņas uz valodas lietotāja līmeni, t.i. dot lietotājam iespēju sapratnes robežās modificēt valodas sintaksi rakstot programmas. Tas arī apskata iespēju, kā var dot valodas lietotājam iespēju izteikt kādu tam nepieciešamu funkcionalitāti ar netriviālām sintaktiskām konstrukcijām, nevis ar funkcijām. Tā ļaus papildināt valodas sintaksi ar ierobežotas makro konstrukciju kopas palīdzību, kas tiek veidota

[1] Sintaktiskais cukurs (*syntactic sugar*) ir speciālas konstrukcijas, kas tiek pievienotas lai to padarītu saprotamāku un lasāmāku cilvēkam. Šīs konstrukcijas nemaina valodas funkcionalitāti, bet gan atvieglo tās lietošanu. Izplatīts sintaktiskā cukura piemērs ir C valodas konstrukcija `a[i]`, kas patiesībā ir `*(a + i)`.

tā, lai ieviestās valodas konstrukcijas joprojām būtu jēdzīgas sākotnējās gramatikas ietvaros.

Darbs piedāvā risinājuma koncepciju, kas ļaus LL(k) parsējamām valodām pievienot sintakses transformācijas sistēmu, kura ļaus paplašināt valodas iespējas un specificēt valodas konstrukcijas pielietošanas domēnam. Tā tiek projektēta tādā veidā, lai tās integrēšana ar kādu eksistējošu parsētāju pieprasītu pēc iespējas minimālas pūles.

Projektētā sistēma sastāv no trim globālām komponentēm - šablonu sakrišanu meklēšanas apakšsistēma, pārrakstīšanas apakšsistēma un tipu apakšsistēma. Šablonu apakšsistēma atpazīst makro šablonu sastapšanas reizes programmas tekstā, pārrakstīšanas apakšsistēma transformē atpazītās virknes, bet tipu apakšsistēma kontrolē pārveidojumu korektību.

Šīs darbs ir fokusēts uz šablonu sakrišanu apakšsistēmas projektēšanas un koncepcijas izstrādes. Transformācijas sistēmas makro šablonu kopa ir paplašināta ar regulāro izteiksmju elementiem, kas ļaus tai būt ekspresīvākai konstrukciju meklēšana. Šī darba ietvaros tika izpētīta un sastādīta algoritmu kopa, kas varētu ļaut efektīvi apstrādāt makro un meklēt transformējamās virknes. Darba gaitā tika izstrādāts prototips, kas parāda specifiskās šablonu sistēmas iespējamību un izveido bāzi tālākai transformāciju sistēmas izstrādei.

Šī dokumenta organizācija ir sekojoša. Nodaļa 2. apraksta šī darba izstrādes pamatojumu un apskata gadījumus, kurus nevar apstrādāt lietojot jau eksistējošos rīkus. Nodaļa 3. apraksta piedāvāto sistēmu, tās īpašības un darbības principus. Nodaļa 4. stāsta par transformācijas sistēmas šablonu apstrādes apakšsistēmas prototipu, par algoritmiem, kas tika lietoti tā izstrādē. Nodaļa 6. apraksta prototipa testēšanas stratēģijas un izstrādes rezultātus, bet nodaļa 7. piedāvā darba secinājumus un projekta turpināšanas perspektīvas.

2. Ievads

Mūsdienu programmēšanas valodas ir spēcīgi rīki, kas var tikt pielietojami dažādu uzdevumu risināšanai. Abstrakti programmēšanas valodas var sadalīt divās grupās - universālas un domēn-specifiskas valodas. Universālas valodas ir pielietojamas visās sfērās, bet domēn-specifiskas gan tiek veidotas kādas konkrētas uzdevumu klases risināšanai. Kaut arī universālas valodas var tikt lietotas šīs uzdevumu klases risināšanai, tās ne vienmēr ir tikpat izteiksmīgas.

Taču parasti tieši universālas programmēšanas valodas tiek lietotas lai risinātu specifiskus uzdevumus, jo ne visiem uzdevumiem eksistē domēn-specifiskas valodas. Domēn-specifiskas valodas ir rīks, ko ir vērts izstrādāt veselai klasei specializētu uzdevumu. Bet to izveide ir liels darbs, ko nav vērts pildīt, lai atrisināt vienu konkrētu uzdevumu.

Gandrīz jebkādas universālas programmēšanas valodas virzošs spēks ir funkcijas un to kompozīcijas. Katra valoda iekļauj sevī kaut kādu bāzes funkcionalitāti, funkciju ietvaru, kuru kompozīcijas un virknes veido programmas. Taču specifisku uzdevumu gadījumā funkciju kompozīcijas nepietiekami apraksta izpildāmo darbību. Dažas darbības ir ērtāk un saprotamāk (tas var būt pieņemts ārpus programmēšanas, reālās dzīves situācijās) attēlot ar citu sintaktisku formu, nevis ar funkciju. Šādas darbības piemērs varētu būt faktoriāla pieraksts - `n!`, kas ir vispārpieņemts apzīmējums.

Viena no šādām sintaktiskām formām ir operatori, tomēr operatoru skaits un pārslogošanas iespējas valodās ir diezgan ierobežotas. To pārslogošana, piemēram, C vidē, neļaus attēlot faktoriālu ar `!` zīmi, jo pārslogojot saglabājas tā prefiksā forma. Tātad operatoru pārslogošana nedod brīvību veidot konstrukcijas, kas varētu paaugstināt valodas izteiksmību. Bet citu rīku, kas varētu to piedāvāt, arī nav, jo programmēšanas valodas bieži iekļauj ļoti ierobežotas sintakses mainīšanas iespējas, vai arī neiekļauj šādu iespēju vispār.

Uz doto brīdi vairākumam mūsdienīgo programmēšanas valodu eksistē standarti. Pateicoties šim faktam, ir iespējams izstrādāt dažādus kompilatorus vienai un tai pašai valodai. Bet sintakses izmaiņu ieviešanas gadījumā ir jāmaina standarts, tātad ir jākorigē arī visi valodas kompilatori. Dažreiz šīs izmaiņas ir tik nopietnas, ka tiešām prasa ievērojamas parsētāja modifikācijas, lai tiktu atbalstītas, bet dažreiz tās ir tikai sintaktiskas, piem. sintaktiskā cukura[1] ieviešana.

Šīs darbs izpēta jautājumu, kā dot valodas lietotājam iespēju izteikt kādu tam nepieciešamu funkcionalitāti ar netriviālām sintaktiskām konstrukcijām, nevis ar funkcijām. Tas arī mēģina iznest valodas sintakses izmaiņas uz valodas lietotāja līmeni, t.i. dot lietotājam iespēju saprātnes robežās modificēt valodas sintaksi rakstot programmas. Tas dos iespēju lietotājiem pašiem ieviest izmaiņas, kas ir nepieciešamas, lai paplašināt valodas iespējas un specificēt valodas konstrukcijas pielietošanas domēnam.

Viena no metodēm, kā varētu ļaut lietotājam paplašināt valodas sintaksi ir izveidot krosskompilatoru, kas transformētu jauno sintaksi tā, lai standarta kompilators to varētu atpazīt. Bet

[1] Sintaktiskais cukurs (*syntactic sugar*) ir speciālas konstrukcijas, kas tiek pievienotas lai to padarītu saprotamāku un lasāmāku cilvēkam. Šīs konstrukcijas nemaina valodas funkcionalitāti, bet gan atvieglo tās lietošanu. Izplatīts sintaktiskā cukura piemērs ir C valodas konstrukcija `a[i]`, kas patiesībā ir `*(a + i)`.

šīs metodes problēma ir tas, ka lielākas daļas moderno valodu sintaksi ir neiespējams noparsēt lietojot automātiskos rīkus. Zemāk ir piedāvāti daži piemēri gadījumiem no populāras valodas C, kad automātiskā parsēšana ir neiespējama.

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

2. C valodā eksistē operators postfikss operators `++`, kas palielina argumentu par vienu vienību. Pieņemsim, ka ir iespēja paplašināt C valodas sintaksi ar infiksu operatoru `++` un pierakstīt konstanšu masīvus `[1, 2, 3]` veidā. Tad izteiksme `a ++ [1]` būtu nepārsējama, jo eksistē vismaz divi to interpretācijas veidi. Tas varētu tikt saprasts ka postfiksā operatora `++` pielietošana mainīgam `a` un tad `a` indeksēšana ar `[1]`. Vai arī tas varētu būt divu masīvu `a` un `[1]` konkatenācija.

Apskatītie piemēri dod iespēju secināt, ka automātisku parsētāju ģeneratoru lietošana var būt tik pat sarežģīta, ka parsētāja rakstīšana ar rokām. Izrādās, ka daudzām eksistējošām valodām parsētāji arī tiek rakstīti manuāli (piemēram C/C++/ObjectiveC kompilators GNU GCC [5]). Tas nozīmē, ka kross-kompilatoru visticamāk būs jāraksta manuāli, risinot eksistējošās gramatikas konfliktus, un oriģinālvalodas ievērojamu izmaiņu gadījumā būs jāpastrādā abi kompilatori, kas nozīmē divreiz vairāk darba.

Šīs darbs apskata pieeju, kas lietos eksistējošo valodas parsētāju ka pamatu savam darbam un piedāvās likumu kopu, kas ļaus modificēt valodas gramatiku parsēšanas laikā. Taču patvaļīgas gramatikas izmaiņas var novest pie nekontrolējamas valodas evolūcijas. Tāpēc aprakstāmā pieejā tiek piedāvātas ierobežotas izmaņu iespējas, kas tiks kontrolētas ar speciāli izveidotas tipu sistēmas palīdzību.

Sistēma ļaus ieviest jaunas konstrukcijas, konstruējot tās no jau eksistējošām valodas vienībām. Tā transformēs programmas gabalus attiecīgi pierakstītiem likumiem tā, lai valodas sākotnējā gramatika būtu tiem pielietojama. Šīs transformācijas korektumu nodrošinās tipu pārbaudes sistēma.

Ļoti līdzīgu uzdevumu, izņemot tikai transformāciju korektuma pārbaudi, pilda arī vispārējie priekšprocesori. Varbūt ir iespējams izveidot minēto sistēmu lietojot kādu no eksistējošām priekšprocesēšanas sistēmām, pievienojot tai kādas korektuma pārbaudes?

Jebkura priekšprocesora viens no galveniem mērķiem ir vienas elementu virknes aizvietošana ar citu. Virknes vienība var būt atšķirīga atkarībā no priekšprocesora, bet parasti šī vienība ir kādu vienas klases rakstzīmju kopa. Klašu daudzums parasti ir fiksēts (skaitlis, burts, tukšums, u.t.t.). Dažreiz zīmes piederība pie klases ir statiska, ka C priekšprocesorā, dažreiz ir dinamiska, ka, piemēram, TEX, kur par atdalītāju var nodefinēt jebkādu specifisku simbolu. Tad apstrāde ir šo virkņu aizvietošana ar citām izveidotām virknēm.

Svarīgākā problēma šādai teksta apstrādes pieejai ir tas, ka tai neiespējams pārbaudīt korektumu. Apskatīsim sekojošu C makro piemēru:

```
#define foo(x, y) x y
```

Pirmkārt, šādam makro nav iespējams statiski izsecināt rezultātu, jo kaut arī `foo(5, 6)` tiks pārveidots par `5 6`, gan `foo(, 5)`, gan `foo(5, )` tiks pārveidots par `5`. Otrkārt, tā kā komats ir makro daļa, nav iespējams kā pirmo makro argumentu padot virkni `5, 6`. To var izdarīt tikai ievietojot argumentu iekavās, tad `foo((5, 6), 7)`, kas tiks pārveidots par `(5, 6) 7`.

Gadījumā, ja ir nepieciešams saplacināt sarakstu, ir nepieciešams izveidot 2 makro, piemēram:

```
#define first(x, y) x
#define bar(x, y) first x y
```

Tomēr aprakstītais makro strādā tikai gadījumos, kad argumentiem ir pareizs tips. Piemēram, izteiksme `bar((5, 6), x)` tiks pārveidota par `5, x`. Bet izteiksme `bar(5, 6)` tiks pārveidota par `first 5 6`, kaut arī tai vajadzētu izraisīt kļūdu.

Var redzēt, ka vispārīgā gadījumā nav iespējams statiski izveidot nekādus secinājumus, tā kā makro rezultāts ir atkarīgs no argumentiem, kuriem tas ir pielietots. Bet patiesībā arī nekādus dinamiskus secinājumus nav iespējams izveidot, jo apstrādājot tekstu neeksistē korektuma kritēriji.

Tomēr pat neņemot vērā korektuma pierādījumu neiespējamību, makro sistēmai trūkst iespēju, lai izveidot jaunas valodas konstrukcijas. Piemēram, būtu dabiski reprezentēt skaitļa moduli ar pierakstu `|a|`. C priekšprocesors, savukārt, ļauj veidot tikai prefiksa formas funkciju makro un konstanšu makro. Jā arī kāda makro sistēma ļautu izveidot minēto pierakstu, parādītos problēmas gadījumos, kad vienam un tam pašam simbolam eksistē dažas nozīmes, piemēram, ar pierakstu `| (a | b) |`, kam jābūt pārveidotam uz `abs(a | b)`.

Apskatītie piemēri parāda, ka ne kross-kompilēšana, ne programmas teksta priekšprocesēšana vispārpieņemtā veidā neder vēlamā rezultāta sasniegšanai. Par šīs problēmas risinājumu varētu kļūt transformācijas sistēma, kas apstrādā nevis programmas tekstu, bet gan programmas tokenu un pseido-tokenu[1] virknes.

Lai padarītu transformācijas sistēmu vairāk spēcīgu un ļautu atpazīt vispārīgākas virknes, tās makro šabloni tiks paplašināti ar regulāro izteiksmju elementiem,

joprojām saglabājot iespēju kontrolēt transformāciju korektumu.

Šādā sistēma dos iespēju paplašināt valodu lietotāja līmenī, nevis kompilatora līmenī. Tas arī dos lielāku brīvību izmaiņu izstrādē, jo lietotājiem būs iespēja veidot makro bibliotēkas ar jaunām iespējām un izplatīt tās. Tā varēs būt pielāgota dažādām valodām, jo tā strādās ārpus paplašināmās valodas gramatikas.

[1] 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`.

Šī sistēma tiek izstrādāta zinātniskās grupas sastāvā, kurā piedalās cilvēki 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).. Tās ideja ir radusies valodas Eq [1] izstrādes darba gaitā un perspektīvā tiks integrēta ar šīs valodas kompilatoru.

[1] Atrodams tiešsaistē - https://github.com/zayac/eq

3. Transformācijas sistēma

Šī nodaļa piedāvā uzbūves principus sistēmai, kura ļaus lietotājam dinamiski paplašināt programmēšanas valodas iespējas ar makro valodas palīdzību. Šī makro valoda ļaus izveidot jaunas valodas konstrukcijas no jau eksistējošām vienībām. Sistēma ir projektēta ka virsbūve parsētājam un strādās paralēli ar parsētāju, analizējot kodu ar ierakstītiem makro un apstrādājot daļiņu virknes.

Šīs sistēmas mērķis ir piedāvāt iespēju modificēt valodas sintaksi programmas rakstīšanas gaitā, nebojājot jau eksistējošo konstrukciju darbu. Sistēma ieviesīs pašmodificēšanos lietojot konstrukciju aizvietošanu, kas vienlaikus nodrošinās gramatikas modifikācijas un sākotnējās valodas gramatikas 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.

Sistēmas raksturīga īpašība ir tas, ka tā nav piesaistīta konkrētai programmēšanas valodai. Uz doto brīdi tā var tikt pielāgota un integrēta dažādu valodu parsētājos, kuru arhitektūra atbilst dažiem nosacījumiem. Tai nav ierobežojumu pret atbalstāmo sintaksi, jo tā strādās ārpus valodas gramatikas.

Tālāk nodaļa ir organizēta sekojoši. Apakšnodaļa 3.1. iedod ieskatu plānotā sistēmas arhitektūrā. Apakšnodaļa 3.2. vispārīgi pamato izvēlēto transformācijas pieeju. Apakšnodaļa 3.3. apraksta īpašības, kuram jāpiemīt parsētājam, lai tas varētu iekļaut transformācijas sistēmu. Apakšnodaļa 3.4. apraksta un pamato sistēmas makro sintaksi. Apakšnodaļa 3.5. detalizētāk apskata sistēmas sadalījumu uz trim apakšsistēmām un apraksta to sadarbību.

3.1. Sistēmas arhitektūra

3.1.1. Makro ielasīšanas soļi

1. Parsētājs atpazīst makro sākšanos un izsauc transformācijas sistēmu.
2. Transformācijas sistēma ielasa makro.
3. Tipu apakšsistēma pārbauda makro tipu korektumu. Ja tipi ir korekti, turpina uz soli 4.. Ja ielasītā makro tipi nav korekti, turpina uz soli 5.
4. Transformācijas sistēma saglabā makro priekš specifiskas produkcijas, lai tālāk apstrādāt programmu.
5. Transformācijas sistēma parāda kļūdas paziņojumu par to, ka ielasītā makro tips nesakrīt ar iezīmēto tipu.

3.1.2. Makro atpazīšanas soļi

1. Parsētājs ienāk kādā no produkciju atpazīšanas funkcijām un izsauc transformācijas sistēmu.

2. Ja transformācijas sistēmā eksistē makro priekš dotas produkcijas, tā sāk makro atpazīšanas procesu ar soli 2a.. Citādi tā nedara neko un parsētājs turpina darbu ar soli 3..

   (a) Transformāciju sistēma pārbauda, vai tajā vietā daļiņu virknē, uz kuru rāda parsētājs, ir sekvence no daļiņām, kas atbilst kādam no makro šabloniem. Ja šāda sekvence eksistē, turpina uz soli 2b.. Ja šādas sekvences nav, turpina uz soli 3..

   (b) Ielasītā sekvence tiek transformēta attiecīgi makro likumam.

   (c) Ielasītā sekvence daļiņu virknē tiek aizvietota ar transformēto sekvenci un parsētāja pozīcija tiek uzstādīta uz aizvietotās virknes sākumu. Transformācijas sistēma sāk darbu atkal no soļa 2a..

3. Parsētājs atjauno darbu no tās pašas vietas daļiņu virknē un sāk gramatikas produkcijas atpazīšanu.

Attēls 3.1. parāda transformācijas sistēmas palaišanu un darbību virkni diagrammas veidā.

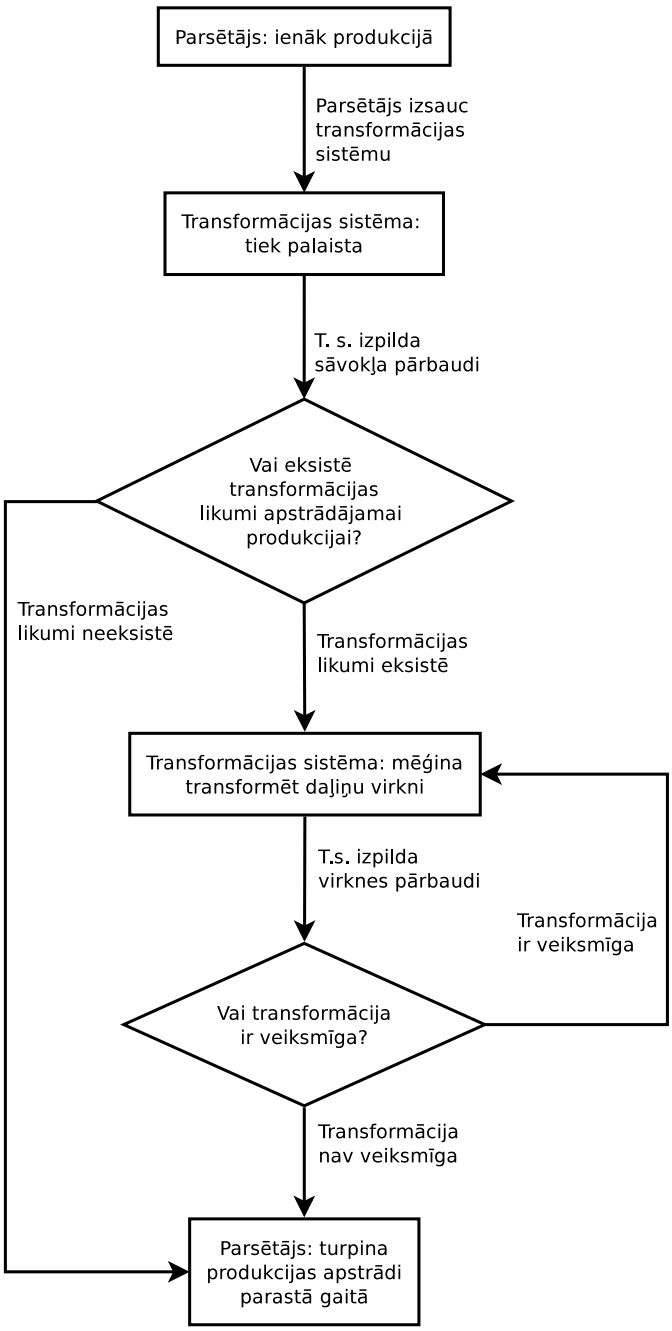
3.2. Transformācijas pieeja

Šī sistēma tiek projektēta ņemot vērā divu eksistējošo pieeju pieredzi. Pirmā no pieejam ir programmas koda priekšprocesēšana - koda makro ierakstu apstrāde pirms parsētāja darba sākšanos. Priekšprocesors parasti aizvieto kādas konstrukcijas ar citām noteikti definētam konstrukcijām. Otrā pieeja ir adaptīvās gramatikas - gramatikas, kas ļauj programmas kodam modificēt savu apstrādes gramatiku. Abas pieejas dod ļoti spēcīgus rīkus programmēšanas valodu izstrādē. Tomēr abām šīm pieejām ir savas problēmas un trūkumi, kurus šīs sistēmas projektēšanā mēģināja atrisināt.

Pirmā problēma, no kuras šī sistēmas izstrādē mēģināja izvairīties ir nekorekta simbolu ar divām nozīmēm apstrāde. Pieņemsim, ka mēs gribam funkciju `abs (x)` apzīmēt ka `| x |`. Apskatīsim izteiksmi `| (a | b) + c|`, kurai vajadzētu tikt pārveidotai par `abs ((a | b) + c)`. Gadījumā, ja transformācijas sistēma apstrādātu tekstu, tā nebūs spējīga pārveidot šādu konstrukciju. Tiks apstrādāti pirmie divi `|`simboli, no `| (a |`, izveidojot nekorektu konstrukciju `abs( (a) b ) + c) |`. Šīs problēmas izvēlētais risinājums ir aprakstīts apakšnodaļā 3.2.1..

Otrā problēma ir dinamisku gramatiku nekontrolējamība. Dinamiskas gramatikas ir ļoti spēcīgs rīks, kas mūsdienās gandrīz netiek lietots. Tas tā ir tādēļ, ka dodot iespēju lietotājam patvaļīgi pievienot un dzēst gramatikas likumus, tiek zaudēta iespēja kontrolēt izmaiņu korektumu. Robežgadījums varētu būt tad, kad sākotnējā gramatika tiek pilnībā aizvietota ar citu gramatiku. Tad, kaut arī sākotnējā gramatika bija derīga parsēšanai ar eksistējošo algoritmu, jaunai gramatikai var piemīt īpašības, kas neļaus to apstrādāt (piemēram, kreisā rekursija LL parsētāju gadījumā). Šīs problēmas izvēlētais risinājums ir apskatīts apakšnodaļā 3.2.2..

Trešā problēma ir tas, ka nav iespējams vienkārši ieviest gramatikas modifikācijas jau eksistējošā valodas parsētāja, ja vien tā arhitektūra no sākuma atbalsta gramatikas izmaiņas. Bet tā kā parasti šāda iespēja netiek iekļauta, visticamāk būs nepieciešamas nopietnas parsētāja adaptācijas. Aprakstāmā sistēma, savukārt, mēģina dot iespēju paplašināt valodas gramatiku



3.1. att. **Parsēšanas darba gaita ar iekļauto transformācijas sistēmu**

bāzējoties uz vienu no plaši lietojamam parsētāju arhitektūrām. Kā tas tiks darīts ir aprakstīts apakšnodaļā 3.2.3..

3.2.1. Priekšprocesori

Ir dažādas pieejas programmu pirmkoda priekšprocesēšanai. Visvairāk izplatītas no tām ir divas pieejas. Viena no pieejām ir sintaktiskā pieeja - sintaktiskie priekšprocesori tiek palaisti pēc parsētāja darbības un apstrādā sintaktiskos kokus, ko uzbūvē parsētājs. Dēļ aprakstāmās sistēmas īpašībām šajā darbā netiks apskatīti sintaktiskie priekšprocesori, jo līdz sistēmas darba izpildei, gadījumā, ja tika ieviestas kaut kādas transformācijas, parsētājs nevar uzbūvēt pareizu sintaktisko koku. Otra no pieejām ir leksiskā, leksiskie priekšprocesori tiek palaisti pirms

pirmkoda parsēšanas un tiek nav zināšanu 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. Apskatīsim jau minētu piemēru `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[1].

Priekšprocesoru var iemācīt apstrādāt šāda veida konstrukcijas un atpazīt tos, ka atomārās izteiksmes. Bet tas nozīmēs, ka priekšprocesoram būs jāzina apstrādājamas valodas gramatika, kas neatbilst priekšprocesora lomai kompilēšanas procesā un nozīmē ka būs divreiz jāimplementē sintakses atpazīšana.

Lai izvairīties no šīs problēmas tika izvēlēts apstrādāt nevis programmas tekstu, bet gan programmas daļiņu un pseido-daļiņu virkni, ko daļēji jau apstrādāja parsētājs. Tas nozīmē, ka makro šablonu sintakse būs bāzēta uz daļiņu aprakstiem, nevis uz tekstuālām izteiksmēm. Piemēram, ja ir nepieciešams izveidot šablonu, kas pārveidos funkcijas ar nosaukumu `bar` par funkcijām ar nosaukumu `foo`, makro šablonā būs jāieraksta daļiņa ar tipu `id` un vērtību `bar` - `{id:bar}`. Tad, kad programmas daļiņu virknē tiks atrasta daļiņa ar šādu tipu un vērtību, tā tiks aizvietota ar citu daļiņu `{id:foo}`. Šāda pieeja dod iespēju programmas tekstā meklēt specifiskus daļiņu tipus, nevis specifiskas teksta daļas. Tas dod iespēju meklēt, piemēram, jebkādus identifikatorus, šablonā norādot daļiņu `{id}` bez vērtības.

Makro šablonu sistēma arī ļauj lietot pseido-daļiņas savu šablonu aprakstos, t.i. ļauj lietot daļiņu `{expr}`. Pseido-daļiņa `{expr}` dotajā sintaksē apzīmē kādu izteiksmi. Tā tiek saukta par pseido-daļiņu tādēļ, ka programmas tekstā tā ir reprezentēta ar citu daļiņu virkni, kaut arī patiesībā tā ir atomāra vienība.

Tā kā aprakstāmā sistēma tiek izstrādāta tā, lai tā varētu darboties nezinot neko par valodas gramatiku, tā nezina, kādas varētu būt izteiksmes dotajā valodā. Tad, kad transformācijas sistēmu makro ir atrodama šāda pseido-daļiņa, sistēma nemēģina pati izsecināt, vai šāda daļiņa ir nolasāma no parsētās virknes. Lietojot speciālu saskarni tā "pajautā" parsētājam, vai sagaidītā pseido-daļiņa ir atrodama sākot no dotās daļiņas virknē. Gadījumā, ja parsētājs var izveidot izteiksmi, tas paziņo par to, un transformācijas sistēma turpina virknes apstrādi. Piemēram, konstrukciju `(a | b) + c` parsētājs atpazīs ka pseido-daļiņu `{expr}`.

Gadījumā, ja kāds no šabloniem tiks atpazīts ieejas virknē, atpazītā apakšvirkne tiks pārveidota citā apakšvirknē, kas aizvietos iepriekšējo. Tālāk aizvietotā virkne tiks apstrādāta ar sākotnējo valodas gramatiku.

Otrā leksiskā tipa priekšprocesoru problēma ir tas, ka tie strādā ārpus programmas tvērumiem. Tas nozīmē, ka tvēruma sākuma daļiņa (piemēram, figūriekava, C/C++, Java un citu valodu gadījumā) tiek uzskatīts par parastu tekstu un var tikt pārrakstīts. Loģiskāk būtu, ja kon-

[1] C/C++ priekšprocesors vispār neatļaus tādu konstrukciju izveidot, kaut arī šāds pieraksts ir diezgan loģisks no matemātiķu skatu punkta. C/C++ makro sistēma ļauj veidot tikai makro konstantes un prefiksa formas funkcijas.

krētā 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.

3.2.2. Dinamiskas gramatikas

Sistēma adaptē dinamisko gramatiku principu, ieviešot izmaiņu kontroli. Dinamiskas gramatikas vispārīgā gadījumā nekontrolē ieviestās izmaiņas, kas var sabojāt valodas parsējamību.

Transformācijas sistēmas makro nepievieno sistēmai jaunus gramatikas likumus tajā nozīmē, ka tie modificē gramatiku. Tie pievieno jaunu daļiņu virknes pārrakstīšanas likumu, kas tiek izpildīts, kad tiek atrasta sakrišana ar makro specificētu šablonu. Tātad jaunās konstrukcijas daļiņu virknē ir atpazītas un pārrakstītas uz konstrukcijām, kuras jau ir zināmas parsētājam. Makro sistēma nedos iespēju dzēst eksistējošos likumus no valodas gramatikas.

Lai nodrošinātu gramatikas likumu pievienošanas kontroli, tiek ieviestas dažas tipu specifikācijas. Katrs no makro attiecas uz kādu konkrētu gramatikas produkciju, un nevar tikt pārbaudīts citā parsēšanas brīdī. Katram makro pārrakstīšanas likumam arī ir specificēts tips, kas tiek lietots lai statiski pārliecināties par to, ka transformācija ir korekta dotā tipa ietvaros. Tipu sistēma detalizētāk ir aprakstīta apakšnodaļā 3.5.3..

3.2.3. Parsētāja modifikācijas

Vēl viena svarīga dinamisku gramatiku īpašība ir tas, ka to lietošanai ir nepieciešams speciāli izstrādāts parsētājs, kas atļauj savu parsēšanas tabulu modifikācijas. Ir jāeksistē iespējai pievienot un dzēst attiecīgus gramatikas likumus, kā arī parsētājam jāprot pārbūvēt parsēšanas tabulas atbilstoši ieviestām izmaiņām.

Šī dinamisku gramatiku īpašība ļoti ierobežo iespēju lietot tos jau eksistējošo valodu paplašināšanai. Šāda veida papildināšana vairākumā gadījumu nozīmēs jauna parsētāja izveidošanu. Bet zinot, ka mūsdienīgām valodām parasti eksistē vairāki kompilatori, kurus izstrādā dažādi cilvēki, šādu izmaiņu ieviešana globālajā līmenī var kļūt neiespējama.

Aprakstāmā sistēma, savukārt, ir domāta ka palīgrīks parsētājam. Parsētāju izmaiņas, kas būs nepieciešamas integrēšanai ar sistēmu ir minimālas. Tas dos iespēju lietot to jebkuram parsētājam, kas atbilst dažiem apstrādes nosacījumiem, kas ir apskatīti apakšnodaļā 3.3..

Sistēma dos iespēju papildināt valodas sintaksi bez nepieciešamības pilnībā pārstrādāt valodas parsētājus. Tā strādās ārpus valodas gramatikas. Pirms katras produkcijas apstrādes ar standartu valodas gramatiku likumiem tiek izsaukta transformācijas sistēma.

3.3. Parsētāju īpašības

Šajā darbā piedāvātā sistēma tiek izstrādāta uz LL(k) parsētāja bāzes. Lai parsētājs varētu tikt integrēts ar aprakstāmo sistēmu, tam jābūt implementētam ar rekursīvas nokāpšanas

algoritmiem LL(k) vai LL(*). LL parsētājs tika izvēlēts tāpēc, ka 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.

Tā kā transformāciju sistēma tiek veidota ka paplašinājums parsētājam, parsētājam jāatbilst dažiem nosacījumiem, kas ļaus sistēmai darboties. Zemāk ir aprakstītas īpašības, kuram jāpiemīt parsētājam, lai tas varētu veiksmīgi sadarboties ar 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. Šīs funkcijas atkārto gramatikas struktūru, tas ir ja gramatikas produkcija A ir atkarīga no produkcijas B, A-vadīšanas funkcija izsauks B-vadīšanas funkciju.

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

Ja izstrādājamās valodas parsētāja modelis atbilst aprakstītām īpašībām, tad tā var tikt veiksmīgi savienota ar aprakstāmo transformācijas sistēmu un ļaut programmētājam ieviest modifikācijas oriģinālās valodas sintaksē.

3.4. 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 korektumu 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 virkni, kurai šī transformācija ir pielietojama. Makro labā pusē ir atrodamas funkcijas, kas izpilda transformē tās tokenu virknes, 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 attēlā 3.2..

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

3.2. 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 tiek aizstāta ar pseido-tokenu virkni, ko atgriezīs `f(v)` un tālāk var tikt reducēta pēc gramatikas produkcijas `prod2` likumiem.

Regulārā izteiksme `regexp` ir vienkārša standarta regulārā izteiksme, kuras gramatika ir definēta attēlā 3.3..

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

3.3. att. **Regulāro izteiksmju gramatika uz pseido-tokeniem**

Pagaidām sistēmas prototipa izstrādē tiek lietota šāda minimāla sintakse, bet nepieciešamības gadījumā tā 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 3.4. izdarītu šo substitūciju, ļaujot programmētājam lietot ērtāku funkcijas pierakstu.

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

3.4. 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 3.5., izpildīs nepieciešamu darbību.

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

3.5. att. **Makro piemērs #2**

3.5. Transformācijas sistēmas apakšsistēmas

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 atrasto virkņu apstrādes 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 var būt derīgs valodas gramatikas ietvaros. Šīs sadalījums ir tikai konceptuāls, kas ir izveidots ērtības dēļ, lai varētu apskatīt sistēmu ka atsevišķu apakšsistēmu kombināciju.

Šablonu sakrišanu meklēšanas sistēma ir īsumā aprakstīta apakšnodaļā 3.5.1.. Sīkāk šīs apakšsistēmas īpašības un tās prototipa realizācija ir aprakstīta nodaļā 4..

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 kāds papildus rīks, par kuru ies runa apakšnodaļā 3.5.2..

Ir plānots, ka transformāciju sistēma varēs atpazīt nepareizi sastādītus makro šablonus lietojot tipu kontroli. Šīs pieejas bāzes principi ir aprakstīti apakšnodaļā 3.5.3.. Jāņem vērā tas, ka lai šī sistēma varētu tikt pielietota, izvēlētai transformāciju valodai jāpiemīt tipu secināšanas (*type inference*) īpašībai.

3.5.1. Sakrišanu meklēšana

FIXME: *Uzrakstīt!*

3.5.2. Tokenu virkņu apstrāde

Gadījumā, ja šablonu sistēma nesaturētu regulāro izteiksmju sintaksi (it īpaši $*$), būtu iespējams transformēt tokenu virknes ar pašu makro palīdzību. Atrastās tokenu virknes vienmēr būtu ar vienādu un determinētu garumu un saturu. Bet tā kā šabloni ļauj meklēt sakrišanas ar elementu sarakstiem, ir nepieciešams veids, kā apstrādāt jaunizveidotus un, iespējams, tukšus sarakstus.

Tātad lai varētu izpildīt atrastās tokenu virknes apstrādi un modificēšanu ir nepieciešams kaut kāds papildus rīks. Šīs rīks varētu būt kaut kāda programmēšanas valoda. Izplatītākās programmēšanas valodu paradigmas mūsdienās ir imperatīvā vai funkcionālā paradigma. Katru no tiem varētu lietot atrasto virkņu apstrādei.

Šīm uzdevumam varētu lietot kādu no imperatīvam programmēšanas valodām, piemēram C, vienkārši izveidojot saskarni ar tās valodas kompilatoru. Bet vairākumam šādu valodu nav statiskas tipu secināšanas iespējas. Statiska tipu secināšana C valodas gadījumā ir neiespējama rādītāju mainīgo dēļ, kuru tipus nevar izrēķināt parsēšanas laikā. Lai varētu ieviest statiskās tipu

izsecināšanas iespējas, vajadzēs ierobežot valodas iespējas, tātad modificēt eksistējošo kompilatoru vai kaut kā citādāk ierobežot pieejamo konstrukciju kopu.

Uzdevumam arī varētu lietot kādu no jau eksistējošām funkcionālām valodām ar tipu secināšanas īpašību. Tad nebūs nepieciešamības veidot savu valodu pilnīgi no jauna. Bet tas nozīmēs, ka būs rūpīgi jāizpēta izvēlētās valodas sintaksi, kas var būt pārāk grūti.

Varētu lietot arī vienu no jau eksistējošām funkcionālām valodām ar tipu secināšanas īpašību, kas piemīt vairākumam funkcionālo valodu.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 izveidot minimalistisku 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 korektumu. Funkcionālā pieeja nodrošina arī to, ka apstrādes funkcijām nepiemīt blakusefekti, kas varētu samainīt eksistējošos datus. Valodas mērķis ir ļaut izveidot atrasto tokenu permutācijas ar kādiem papildinājumiem nepieciešamības gadījumā.

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.

3.5.3. Tipu sistēma

Kā bija redzams attēlā 3.2., 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[1]. 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. Kad šablona apstrāde tiek sākta, var arī izsecināt iespējamo

[1] Sk. nodaļu 4.6..

virknes garumu intervālu un izvadīt brīdinājumus gadījumā, ja ir iespējama funkcijas $head(x)$ izsaukšana no tukšā saraksta.

Šī nodaļa tikai vispārīgi apraksta tipu secināšanas sistēmas ideju. Uz doto brīdi tā atrodas izstrādes stadijā, bet var redzēt ka tās izveide ir pamatota. Sīkāka informācija par tipu sistēmu ir saņemama pie Eq kompilatora izstrādes komandas.

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.

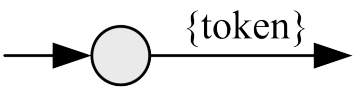
4.6.1. Regulāro izteiksmju pārveidošana nedeterminētā galīgā automātā

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.

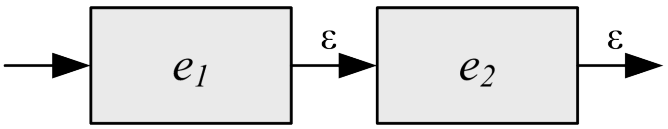
Nedeterminēts galīgs automāts (NGA) veselai regulārai izteiksmei ir izveidots to daļējiem automātiem katrai regulārās izteiksmes daļai. Katram operatoram tiek piekārtota attiecīga konstrukcija. Daļējiem automātiem nav akceptējošu stāvokļu, tiem ir pārejas uz nekurieni, kuras vēlāk tiks lietotas lai savienotu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

Attēlā 4.6. ir parādīts NGA vienam tokenam $token$.



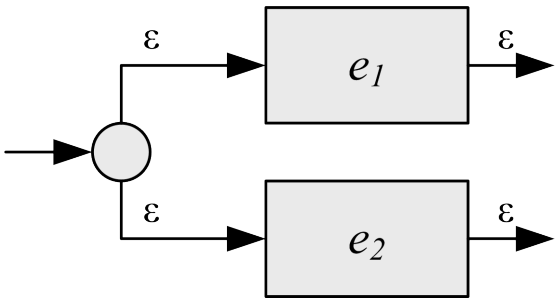
4.6. att. **Automāts vienam tokenam**

Attēlā 4.7. ir parādīts NGA divu automātu konkatenācijai $e_1 e_2$.



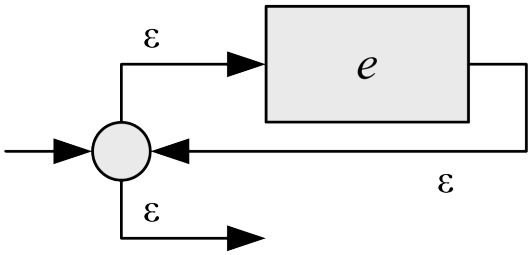
4.7. att. **Automāts divu automātu konkatenācijai**

Attēlā 4.8. ir parādīts NGA izvēlei starp diviem automātiem $e_1|e_2$.



4.8. att. **Automāts izvēlei starp diviem automātiem**

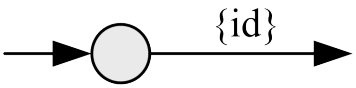
Attēlā 4.9. ir parādīts NGA priekš konstrukcijas e_1*.



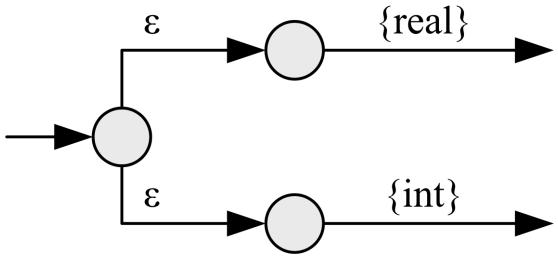
4.9. att. **Automats automātu virknei**

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

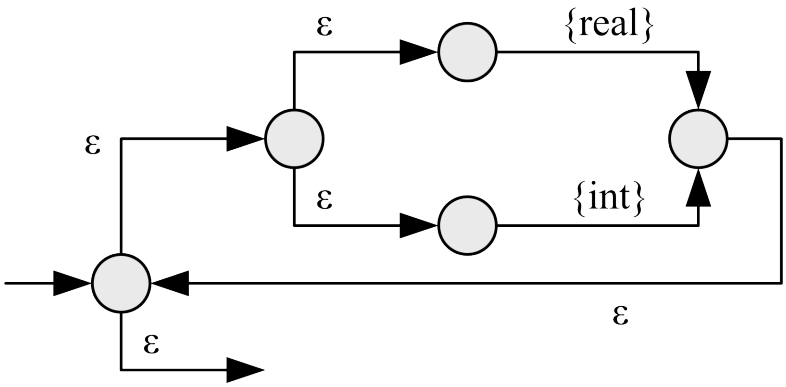
Apskatīsim piemēru - izteiksmi `{id} ({real} | {int}) *`. Sākumā tiek izveidoti NGA izteiksmes daļām. Attēls 4.10. parāda NGA priekš `{id}`. Attēls 4.11. parāda NGA priekš daļas `{real} | {int}`. Attēls 4.12. parāda NGA priekš `({real} | {int}) *`.



4.10. att. **Automats tokenam `{id}`**

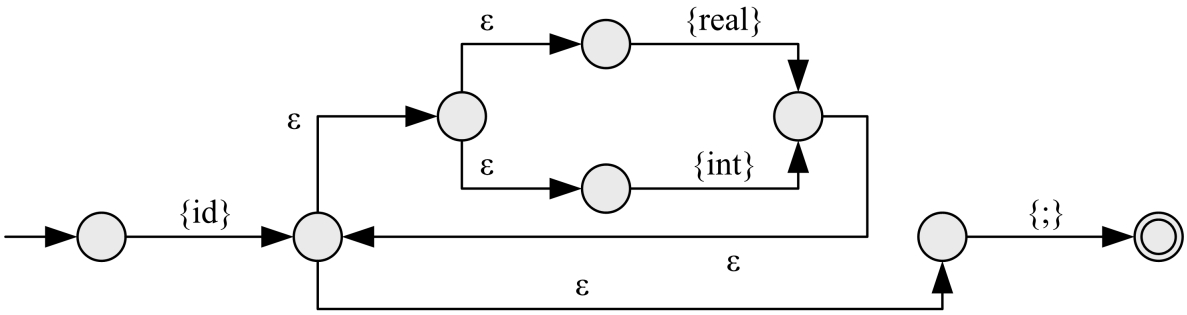


4.11. att. **Automats izteiksmei `{real} | {int}`**



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

Tad izveidotie automāti var tikt savienoti un beigās tiem tiek pievienots akceptējošais stāvoklis (attēls 4.13.).



4.13. 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. [6], 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 4.1. 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.

4.1. 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 4.14. 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, 6]

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

4.14. 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] [6] 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 4.15. 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
$\Pi_{final} = \Pi$

4.15. 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.

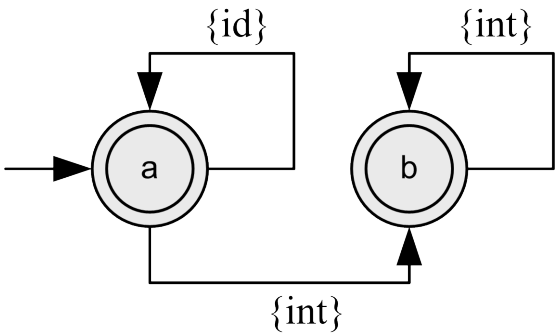
Algoritms 3: Divu DGA apvienošana

Ieeja: DGA D_1 un D_2.

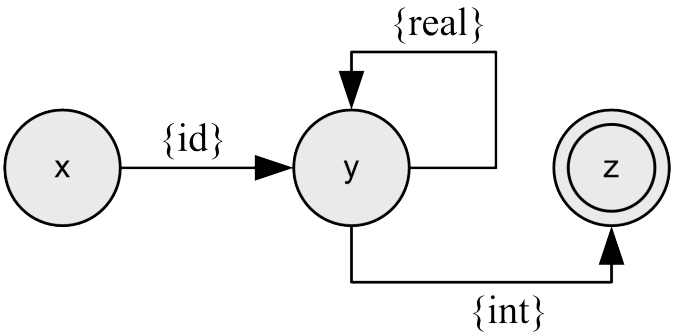
Izeja: Jauns DGA D', kas apvieno D_1 un D_2.

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 4.16. parāda determinētu minimālu automātu priekš regulārās izteiksmes `{id}* {int}*`. Tas satur divus stāvokļus, `a` un `b`, kuri abi ir akceptējoši. Attēls 4.17., savukārt, parāda DGA priekš izteiksmes `{id} {real}* {int}`.

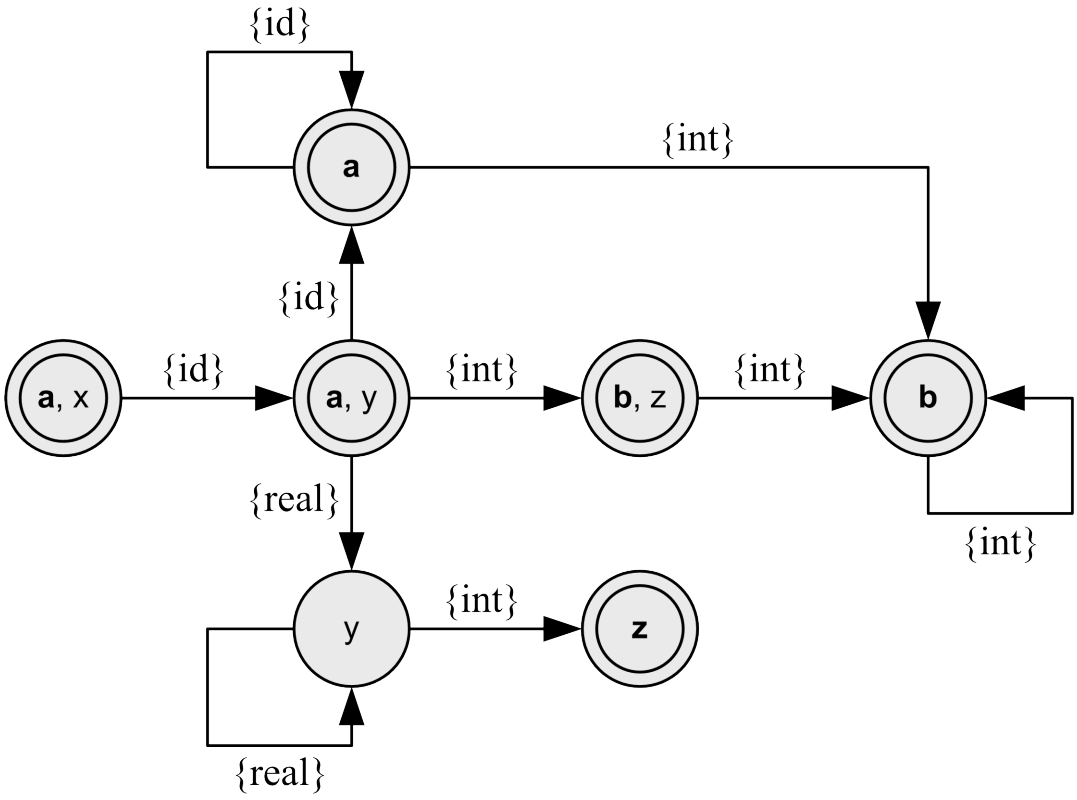


4.16. att. **Automats izteiksmei `{id}* {int}*`**



4.17. att. **Automats izteiksmei `{id} {real}* {int}`**

To apvienotais automāts ir parādīts attēlā 4.18..



4.18. att. **Automats izteiksmju `{id}* {int}*` un `{id} {real}* {int}` apvienojumam**

Attēls 4.19. 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

4.19. 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 [8].

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. [8]

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

Šīs darbs tika iedvesmots ar dažiem rakstiem par dinamisko gramatiku iespējām un pielietojumiem programmēšanas valodu izstrādē. Apakšnodaļa 5.1. apskata darbus par dinamiskām gramatikām.

Vispārīgi dinamiskas gramatikas un valodu dinamiska parsēšana gandrīz netiek lietota valodu implementācijās. Tomēr valodu paplašināšana ir zināms uzdevums, kuram eksistē dažādi risinājumi. Katrs no risinājumiem ir darbspējīgs un pamatots priekš sava mērķa, un katram ir savas labās un sliktās puses. Šī nodaļas apakšnodaļas 5.2., 5.3., 5.4. un 5.5. piedāvā līdzīgu projektu un darbu apskatu, kā arī uzrāda aprakstītā projekta atšķirības no šiem projektiem.

Šī nodaļa neiedziļinās sistēmu sintakses īpatnībās, jo šāda apskate būtu pārāk apjomīga. Tā tikai pavirši apskata nozīmīgākas sistēmu īpatnības. Tālākai izpētei katra apakšnodaļa piedāvā literatūras avotus, kas piedāvā nepieciešamu informāciju.

5.1. Dinamiskas gramatikas

Ir dažas dinamisku gramatiku pieejas, kas, diemžēl, vairākumā ir tīri teorētiskas. Labu ieskatu adaptīvo gramatiku pieejās dod Heninga Kristiansena raksts [**?**] un Džona Šutta maģistra darbs [**?**]. Abi šie darbi apkopo visas uz to brīdi eksistējošās pieejas. Diemžēl kopš abu raksta laika citu ievērojamu variantu un implementāciju skaits ir ļoti mazs.

Pjērs Bulliers savā rakstā [**?**] apskata iespēju lietot dinamiskas gramatikas valodas sintakses kontrolei. Tas apraksta, kā tās dod iespēju pārbaudīt tipus programmas parsēšanas, nevis kompilēšanas fāzē. Diemžēl, aprakstīta sistēma ir eksperimentāla un tikai prototipēta, nevis izveidota par lietojamu risinājumu.

FIXME: *Rewrite!*

5.2. Lisp

Lisp (*LISt Processing*) ir viena no funkcionālam valodām, kuras ievērojama īpašība ir spēcīga meta-programmēšanas iespēja. Lisp ļauj paplašināt valodas konstrukcijas ar makro izteiksmēm un pievienot valodai jaunus atslēgas vārdus.

Lisp gan dati, gan programmas kods ir attēloti sarakstu veidā, tātad funkcijas var tikt apstrādātas tāpat ka dati. Tas dod iespēju rakstīt programmas, kas manipulē ar citām programmām un iedod bezgalīgas iespējas programmētājam, kuram nav nepieciešamības mācīt jaunu valodu, lai modificētu eksistējošo. Sintakses paplašināšana ir izpildāma lietojot pašu Lisp un tā makro sistēma ļauj veidot Lisp domēn-specifiskus dialektus.

Lisp makro apstrādes spējas ir ļoti specifiskas tieši šai valodai. Tas var tikt lietotas tāpēc, ka pati valoda ir speciālā veidā implementēta un uztver visu informāciju vienādi. Lisp makro sistēma bez izmaiņām nav pielietojama imperatīvām valodām, jo to instrukciju kopa ir cieši atdalīta no programmas datu kopas.

Lisp ļauj pievienot valodai jaunus atslēgas vārdus, bet neļauj veidot jaunus operatorus ne infiksā, ne postfiksā formā. Visām jaunām konstrukcijām joprojām jābūt prefiksa notācijā un to argumentiem saraksta formā.

Visas iegūtās konstrukcijas joprojām būs tīri funkcionālas, ar Lisp-specifisku sintaksi, t.i. nebūs iespējas izveidot moduļa pierakstu `|a|`. Lisp sintakse ir grūti saprotama cilvēkasm, kas nepazīst valodu programmēšanas līmenī, t.i. ja nestrādāja ar to jau iepriekš. Ar Lisp makro sistēmu nav iespējams izveidot sintaksi, kas būtu lasāma un saprotāma cilvēkam kas neprogrammē.

[7]

5.3. Forth

Forth ir steka valoda, kas neatbalsta nekādas programmēšanas paradigmas un vienlaikus atbalsta tās visas. Pateicoties Forth īpatnībām, tā var tikt lietota vienlaikus ka valoda un ka kompilators.

Forth satur tikai divus daļiņu tipus, skaitļus un visus citus - vārdus. Visas valodas vienības ir vārdi, gan funkcijas, gan mainīgie, gan operatori. Šāda pieeja ļauj rakstīt programmas dabiskā valodā. Tā kā tā ir steka valoda, nav nepieciešamības lietot iekavas lai padotu parametrus funkcijām, vai citādi atdalīt funkcijas jēdzienu no mainīgo jēdziena. Forth nesatur nekādus atslēgvārdus, kas tiek apstrādāti atsevišķi. Forth standarts definē vārdu kopu, kas ir iebūvēti valodā, bet arī tie var tikt pārdefinēti.

Visas konstrukcijas ir ieraksti Forth vārdnīcā, ar kuru var manipulēt kā ar datiem. No šī viedokļa Forth ir līdzīgs Lisp, kas arī uztver programmu un datus vienādi. Tas līdzīgi dod iespēju modificēt izpildāmo kodu un paplašināt valodas sintaksi, bez nepieciešamības mācīties jaunu transformācijas valodu.

Forth ļauj ne tikai veidot jaunas sintaktiskas konstrukcijas, bet ļauj arī iejaukties kompilēšanas procesā. Tas tiek atļauts ar speciāliem vārdiem, un ļauj iegulst pat citu valodu kodu Forth programmā. Tomēr interpretēt iegulto kodu vajadzēs pašam programmētājam, kas grib to izpildīt.

Diemžēl Forth īpatnības padara to ļoti specifisku lietošanā. Potsfiksā forma ir diezgan izteiksmīga valodiski[1], bet nav izteiksmīga gadījumos, kad ir nepieciešams ieviest matemātikas notācijas. Tam, ka var rakstīt programmas dabiskā valodā, arī ir divas monētas puses - ja katrs rakstīs savā valodā, citiem programmētājiem visticamāk būs grūti saprast (ja vien vispār tas būs iespējams).

Forth implementēto sistēmu nebūs iespējams pielietot valodās, kuras satur vairākus tokenu tipus, jo nebūs iespējams apstrādāt kodu un datus vienādi. Forth vide ir ļoti vienkārši implementējama un dod ļoti lielas iespējas. Tā kā tā neatbalsta ...

Plašāka informācija par Forth valodu un tās iespējām ir atrodama ...

FIXME: *Rewrite!*

[1] Sakarā ar sintakses īpašībām pēc filmas "Zvaigžņu kari" parādīšanās, The mistery of Yoda's speech uncovered is: Just an old Forth programmer Yoda was.

5.4. Nemerle

Nemerle ir statiski tipizējama universāla programmēšanas valoda .NET platformai. Tai piemīt gan funkcionālas, gan objektorientētas, gan imperativās paradigmas iezīmes. Tai ir C#-līdzīga sintakse un ļoti spēcīga meta-programmēšanas sistēma.

Viena no svarīgākām Nemerle pazīmēm ir tas, ka tai ir raksturīga ļoti augsta līmeņa pieeja visiem valodas aspektiem. Tā mēģina atbrīvot programmētāju no lieka darba lietojot tipu izvadīšanas iespēju un makro sistēmu.

Nemerle makro sistēma dod iespēju ģenerēt tekstveidnes (*boilerplate*) kodu bez programmētāja piepūles. Tas arī dod iespēju kompilatoram izpildīt statiskas pārbaudes kompilācijas laikā. Tas viss dod spēku programmatiski ģenerēt pārbaudāmu kodu.

Kaut arī Nemerle makro sistēma ir ļoti spēcīga un ļauj izpildīt daļēju novērtēšanu, tomēr tā ir ļoti atkarīga no valodas specifikas. Tā kā Nemerle ir statiski tipizēta, tā dod iespēju kompilatoram pārbaudīt kompilēto kodu. Diemžēl plaši lietojamas valodas ne vienmēr ir statiski tipizējamas (piem. C/C++, Python), un šāda pieeja nebūs realizējama vairākumam valodu.

Plašāka informācija par Nemerle valodu un makro sistēmas īpatnībām ir atrodama [**?**].

FIXME: *Rewrite!*

5.5. OpenZz

OpenZz parsētājs ir interpretējams dinamisks parsētājs, kas ļauj ātri izstrādāt parsēšanas risinājumus. OpenZz ļauj modificēt un paplašināt savu gramatiku lietojot komandas parsējamo valodu. To var pielāgot dažādu valodu parsēšanai, bet izstrādāts tas tika lietošanai "Apese" programmēšanas valodai.

Ļoti svarīga šī parsētāja īpašība ir tas, ka tas ļauj modificēt valodas gramatiku ar pašas valodas palīdzību. Tomēr tā kā parsētājs atbalsta parsēšanas tabulu izmaiņas, kas ietekmē parsētāja ātrdarbību.

Tas nav pielietojams jebkādai jau eksistējošai valodai, tam ir jāievieš speciāli modificēšanas mehānismi.

Diemžēl šīs parsētājs netiek attīstīts kopš 2002. gada un informācija par to ir ļoti ierobežota. Sīkāka informācija par šo rīku ir dabūjama rakstā [**?**], kas apskata parsētāja koncepciju, un parsētāja mājaslapa [**?**].

FIXME: *Rewrite!*

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 6.1. parāda konceptuālu testu sadalījumu pa grupām. Dažas grupas pārklājas, jo, piemēram, tvērumu pārbaudošie testi pārbauda arī korektas regulāro izteiksmju prioritātes.

6.1. 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.

6.1. 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.

6.1. 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 sa-

krišanu 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

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] Free Software Foundation. Gcc. http://gcc.gnu.org.

[6] 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.

[7] Peter Seibel. *Practical Common Lisp*. Apress, September 2004.

[8] 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.