LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

AR REGULĀRĀM IZTEIKSMĒM PAPLAŠINĀTU GRAMATIKU DINAMISKA PARSĒŠANA

BAKALAURA DARBS

Autors: **Jūlija Pečerska**
Studenta apliecības Nr.: jp08194
Darba vadītājs: profesors Dr. dat. Guntis Arnicāns

RĪGA, 2012

Anotācija

Mūsdienu programmēšanas valodu lielākai daļai ir statiski definēta nemainīga sintakse un tās nepiedāvā līdzekļus valodas paplašināšanai. Neskatoties uz to, ka šāds projektējums ir pamatots, tas nozīmē, ka visas, pat maznozīmīgas, izmaiņas valodas sintaksē ir nepieciešams implementēt valodas kompilatorā.

Šajā darbā tiek aprakstīts mehānisms, kas ļaus lietotājam paplašināt valodas sintaksi. Mērķa sasniegšanai tiek projektēta sistēma, kas pēc savas būtības ir ļoti līdzīga priekšprocesoram, bet kurai piemīt ciešāka integrācija ar programmēšanas valodas semantiku.

Mehānisms ir balstīts uz regulāro izteiksmju šabloniem, kuri veido saskarni sintakses transformācijām. Šīs darbs koncentrējas uz efektīvas šablonu sakrišanu meklēšanas pieejas izveidošanas, pamatojoties uz kuru vēlāk tiks izstrādāta transformācijas sistēma.
Atslēgvārdi: Dinamiskās gramatikas, priekšprocesēšana, makro, regulārās izteiksmes, galīgi determinēti automāti, Python

Abstract

Dynamic parsing using regular-expression-extended grammars

Most of the programming languages nowadays have statically fixed syntax and do not provide any means of extending it. There are reasons for such s design decision, however, it implies that any, even slight, syntactical modification of the language have to be implemented on the compiler side.

This thesis describes a mechanism that allows syntactical extensions to a languge to be introduced by the user, rather than by the compiler developer. In order to achieve that a system is designed, which is very similar to a preprocessor, but which has a much closer integration with the semantics of the language.

The mechanism is based on regular expression templates, which are the interface to the transformations of the syntax. This paper is focused on creating an effective template matching approach, which later will be used as the basis for building the transformation system.

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

Saturs

APZĪMĒJUMU SARAKSTS

ε-pārejas (angl. *ε-transitions*) Nedeterminētā galīgā automātā pārejas, kas ļauj pārvietoties uz jaunu stāvokļi neielasot nekādu ieejas simbolu.

Abstraktais Sintaktiskais Koks, ASK (angl. *Abstract Syntax Tree, AST*) Abstraktais sintaktiskais koks (ASK) ir pirmkoda sintaktiskās struktūras kokveida reprezentācija. Katrs koka mezgls apzīmē kādu konstrukciju no pirmkoda. Parsētājs apstrādājot kodu veido ASK, kas tālāk tiek kompilēts izpildāmā kodā.

Atpakaļnorāde (angl. *backreference*) Regulārā izteiksmē grupām, kuras veido izteiksmes daļas, kas ir ievietotas iekavās, parasti tiek piekārtotas atpakaļnorādes, kas glabā atbilstošu atrastās virknes daļu.

Daļiņa (angl. *token*) Pirmajā kompilēšanas stadijā leksiskais analizators sadala tekstu leksēmās un katrai leksēmai izveido speciālu objektu, kas tiek saukts par daļiņu. 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}`

Determinēts galīgs automāts (angl. *deterministic finite automaton*) Determinēts galīgs automāts ir automāts, kas akceptē vai neakceptē galīgu elementu virknes un apstrādā tos ar vienu iespējamo ceļu pa automāta stāvokļiem katrai ieejas virknei. [11]

Leksiskais analizators (angl. *lexer*) Programma vai funkcija, kas izpilda leksisko analīzi - ieejas simbolu virknes pārveidošana daļiņu virknē. Bieži sastāv no vienīgas funkcijas, ko izsauc parsētājs, un kas atgriež nākamo daļiņu no ieejas simbolu virknes.

Leksēma (angl. *lexeme*) Leksēma ir nozīmīgo simbolu secība pirmkodā, piemēram, burtu vai ciparu virkne, kas veido vienu elementu ieejas tekstā.

LL-parsētājs (angl. *LL-parser*) Parsētājs kontekstneatkarīgu gramatiku apakškopas parsēšanai, kas analizē ieejas virkni no kreisās puses uz labo un veido kreiso atvasinājumu (angl. *derivation*) gramatikas produkciju reducēšanai.

Meta-programmēšana (angl. *meta-programming*) Meta-programmēšana ir tādu programmu rakstīšana, kas strādā ar savu vai citu programmas izejas kodu kā ar ieejas datiem, vai arī izpilda daļu no darba kompilēšanas laikā, nevis darba laikā.

Nedeterminēts galīgs automāts (angl. *nondeterministic finite automaton*) Nedeterminēts galīgs automāts ir automāts, kas akceptē vai neakceptē galīgu elementu virknes, toties var vienlaikus atrasties dažos stāvokļos. [11]

Parsētājs (angl. *parser*) Programma vai funkcija kas izpilda sintaktisko analīzi - programmas daļiņu virknes sastatīšana attiecībā pret programmēšanas valodas gramatiku. Parsētāja darba rezultāts parasti ir ASK.

Priekšprocesors (angl. *preprocessor*) Priekšprocesora ir programma, kas ielasa datus un izvada tos tādā formā, kurā tie ir piemēroti citas programmas ieejai. Parasti priekšprocesori apstrādā programmu pirmkodu, ko pēc tam apstrādā kompilators. Apstrādes veids un rezultāts ir atkarīgs no priekšprocesora veida, parasti tie ir spējīgi tikai izpildīt teksta apstrādi, kamēr daži pēc iespējām ir līdzīgi programmēšanas valodām.

Pseido-daļiņa (angl. *pseudo-token*) Pseido-daļiņa ir citu daļiņu grupa, kas tiek aizvietota ar vienu objektu. Tas var tikt darīts, lai vienreiz noparsētu izteiksmi nevajadzētu apstrādāt vēlreiz. Daļiņu virkņu aizvietošana ar pseido-daļiņām notiek gramatikas produkcijas reducēšanas brīdī. Kad, piemēram, daļiņu virkne `{id:a} {+} {id:b}` tiek atpazīta ka derīga izteiksme gramatikas ietvaros, tā var tikt aizvietota ar pseido-daļiņu `{expr:a + b}`

Regulārā izteiksme (angl. *regular expression*) Regulārās izteiksmes ir formāla valoda apakšvirkņu meklēšanai. Regulārā izteiksme ir šablons, kas sastāv no simboliem un speciāliem meta-simboliem, kas uzdod meklēšanas kritērijus. Populārākie meta-simboli ir `*`, `+`, `?`, `|`. Pieņemsim, ka `s` un `r` ir regulārās izteiksmes. Tad `s*` meklēs sakrišanu pa izteiksmi `s` 0 vai vairākas reizes, `s+` meklēs `s` vienu vai vairākas reizes, `s?` meklēs `s` 0 vai 1 reizi, bet `s | r` meklēs sakrišanu ar vienu no izteiksmēm `s` vai `r`.

Rekursīvā nokāpšana (angl. *recursive descent*) Pirmkoda parsēšanas metode, kur katras valodas gramatikas produkcijas apstrāde ir realizēta ar parsēšanas funkciju. Funkcijas izsauc viena otru un pakāpeniski no kreisās puses uz labo ielasa pirmkoda daļiņas.

Tipu izsecināšana (angl. *type inference*) Programmēšanas valodās tipu izsecināšana ir iespēja loģiski izvest kādas izteiksmes tipu.

Tvērums (angl. *scope*) 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 tvērumā (globālie mainīgie), var tikt pārdefinēti mazākajā tvērumā (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 tvēruma beigām, nemainot globālā mainīgā vērtību.

IEVADS

Mūsdienu programmēšanas valodas ir spēcīgi rīki, kas var tikt pielietoti dažādu uzdevumu risināšanai. Atkarībā no pielietojuma tās tiek klasificētas universālās un domēn-specifiskās: domēn-specifiskās valodas tiek pielietotas konkrētas uzdevumu klases risināšanai, toties universālas - visiem uzdevumu veidiem. Domēn-specifiskās valodas projektēšana un implementācija ir pietiekami laikietilpīgs darbs, lai to veiktu viena konkrēta uzdevuma risināšanai, līdz ar to šīs valodas projektēšanai ir nozīme tikai veselas specializētu uzdevumu klases gadījumā.

Katras universālas programmēšanas valodas pamata sintaktiskais spēks ir funkcijas un to kompozīcijas. Katrai valodai piemīt noteikta bāzes funkcionalitāte – funkciju ietvars, kur funkciju kompozīcijas un virknes veido atbilstošās programmas. Taču ir iespējamas situācijas, kad funkciju kompozīcijas nepietiekami izteiksmīgi apraksta izpildāmo darbību; kā arī dažas darbības ir ērtāk un saprotamāk attēlot ar citu sintaktisku formu, nevis ar funkciju (piemēram, faktoriālus ar pierakstu `n!`).

Viena no sintaktiskām formām, kas varētu paagustināt koda lasāmību, ir operatori. Tomēr bieži vien programmēšanas valodas stingri ierobežo pieejamo operatoru un to argumentu skaitu, to asociatīvās īpašības (kreisā vai labā asociativitāte) un novietojumu (prefiksa, infiksa, postfiksa)[1]. Neskatoties uz to, ka, piemēram, C++ valoda ļauj pārslogot operatorus, tā nedod iespēju izveidot postfiksu konstrukciju `n!` faktoriāla apzīmēšanai. Šāda uzvedība ir tipiska lielākai programmēšanas valodu daļai.

Vienīga iespēja, kas šajā situācijā ļaus ieviest izmaiņas valodā, ir izmainīt valodas sintaksi. Uz doto brīdi lielākai mūsdienīgo valodu daļai eksistē standarti, kas ļauj izstrādāt dažādus kompilatorus vienai un tai pašai valodai. Sintakses izmaiņu ieviešanas gadījumā ir jāmaina standarts un tātad arī valodas kompilatori. Dažreiz šīs izmaiņas ir tik nopietnas, ka tiešām to atbalstīšanai ir nepieciešamas ievērojamas parsētāja modifikācijas, tomēr dažos gadījumos tās var būt tikai sintaktiskas, piem., sintaktiskā cukura[2] ieviešana.

Gadījumos, kad ir nepieciešams paplašināt valodu tā, lai tā būtu piemērotāka konkrētam uzdevumam, sintakses izmaiņas var nebūt būtiskas, tomēr lielāka valodu daļa arī šādas izmaiņas neļaus ievest bez valodas gramatikas un standarta modifikācijas. Tas ievērojami palēnina iespējas eksperimentēt ar valodas sintaksi.

Balstoties uz šiem novērojumiem, tiek izveidots šīs darbs, kas pēta, vai ir iespējams iznest valodas sintakses izmaiņas uz lietotāja līmeni, t.i. dot valodas lietotājam iespēju modificēt valodas sintaksi programmas rakstīšanas procesā, kā arī ļaut izteikt viņam nepieciešamo funkcionalitāti ar netriviālām sintaktiskām konstrukcijām, nevis izmantojot funkcijas. Darba mērķis ir atļaut lietotāja līmenī papildināt valodas sintaksi ar ierobežotas makro konstrukciju kopas palīdzību, kas ir veidota tā, lai ieviestās valodas konstrukcijas pēc transformēšanas joprojām būtu

[1] Dažas valodas, piemēram, Prolog, ļauj ieviest un pārslogot operatorus un mainīt to asociativitāti un novietojumu, bet neļauj veidot vairāku (>2) argumentu operatorus. Toties ir jāņem vērā, ka Prolog ir eksotiska valoda, pie kuras sintakses ir nepieciešams pierast. [10]

[2] Sintaktiskais cukurs (*syntactic sugar*) ir speciālas konstrukcijas, kas tiek pievienotas, lai padarītu valodu cilvēkam saprotamāku un lasāmāku. 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)`.

jēdzīgas sākotnējas gramatikas ietvaros.

Šīm mērķim tika izvēlēta pieeja, kas ir līdzīga koda priekšprocesēšanai. Programmas teksta priekšprocesēšanas gaitā nav iespējams precīzi paredzēt transformācijas rezultātu[1]. Darba mērķa sasniegšanai tiek piedāvāts makro šablonos lietot izejas koda daļiņas (angl. *token*) un gramatikas produkcijas, kas dos iespēju kontrolēt katra šablona argumenta tipus. Šablonu transformācija tiks iznesta uz citu abstrakcijas līmeni un ļaus apstrādāt virknes ar daļējo izrēķināšanas iespēju.

Darbs piedāvā risinājuma koncepciju, kas ļaus LL(k) parsējamām valodām pievienot sintakses transformācijas sistēmu, kas paplašina valodas iespējas un specificē valodas konstrukcijas pielietošanas domēnam. Projektēta sistēma ir veidota tā, lai tās integrēšana ar kādu eksistējošu parsētāju pieprasītu pēc iespējas minimālas pūles. Sistēma sastāvēs no trim globālām komponentēm: šablonu sakrišanu meklēšana apakšsistēma, pārveidošanas apakšsistēma un tipu apakšsistēma.

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 meklēšanā. Šī darba ietvaros tika izpētīta uzdevuma risināšanas iespējas un tika sastādīta algoritmu kopa, kas atļautu efektīvi apstrādāt makro un meklēt transformējamas virknes. Darba gaitā tika izstrādāts prototips, kas parāda specifiskās šablonu sistēmas iespējamību un izveido pamatu tālākai transformācijas sistēmas izstrādei. Prototipa izstrādes un testēšanas laikā tika identificēti šablonu sistēmas iespējamie ierobežojumi.

Šī dokumenta organizācija ir sekojoša. 1. nodaļa apraksta šī darba izstrādes pamatojumu un apskata gadījumus, kurus nevar apstrādāt, lietojot jau eksistējošos rīkus. 2. nodaļa apraksta piedāvāto sistēmu, tās īpašības un darbības principus. 3. nodaļā tiek apskatīts transformācijas sistēmas šablonu apstrādes apakšsistēmas prototips un darbā pielietotie algoritmi. 4. nodaļa apskata dažus līdzīgus projektus un īsi atspoguļo to saturu. Tālāk tiek aprakstīti darba izstrādes rezultāti, secinājumi un projekta turpināšanas perspektīvas.

[1] Piemēram, gadījumos, kad priekšprocesēšanā tiek lietotas zarošanas konstrukcijas vai ir nepieciešama vairākkārtīga apstrāde, lai dabūtu beigu rezultātu.

1. PROBLEMĀTIKA UN RISINĀJUMA KONCEPCIJA

Šī nodaļa apraksta darba problemātiku un ieskicē izvēlētā risinājuma koncepciju. Sīkāk risinājums tiks apskatīts 2. un 3. nodaļās.

1.1. Idejas izcelsme

Šīs makro transformācijas sistēmas ideja ir radusies valodas Eq[1] kompilatora izstrādes gaitā, ar ko nodarbojas zinātniskā grupa, kuras sastāvā ir 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). Šīs valodas sintakse bāzējas uz LaTeX teksta procesora sintakses, kas ir standarts priekš zinātniskām publikācijām. Korekti uzrakstīta Eq valodas programma var tikt interpretēta ar LaTeX procesoru. Perspektīvā Eq programma varēs tikt kompilēta un izpildīta uz vairākuma mūsdienīgo arhitektūru.

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

Kaut arī ideja un pieejas izstrāde sākās ar valodu Eq, tā tiek projektēta tā, lai varētu tikt lietota dažādām valodām, ne tikai Eq. Tas tiek darīts lai paplašināt iespējamo sistēmas pielietošanas sfēru un tās attīstības iespējas.

1.2. Problemātika

Valodas sintakses modificēšanai lietotāja līmenī ir izveidoti dažādi risinājumi, daži no kuri ir izskatīti 4. nodaļā. Toties katrs no eksistējošiem risinājumiem ir kaut kādā mērā ierobežots savās iespējās. Šī nodaļa parāda projektētās transformācijas sistēmas pieeju un pamato risinājuma koncepcijas izveidi.

Viena no metodēm, kā varētu ļaut lietotājam paplašināt valodas sintaksi ir izveidot kross-kompilatoru, kas transformētu jauno sintaksi tā, lai standarta kompilators to varētu atpazīt. Toties tā kā lielākās moderno programmēšanas valodu daļas sintaksi ir grūti noparsēt lietojot automātiskus rīkus, kross-kompilatora izveidošana var būt problemātiska. 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.

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

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

Dažreiz arī programmatūras koda dalīšana pa daļiņām ir atkarīga no šī koda konteksta, kas padara ne tikai parsēšanas procesu, bet arī leksēšanas procesu neautomatizējamu.

1. Valoda C++ ļauj lietotājam izveidot ligzdveida veidnes, piemēram, šādas `template <typename foo, list <int>>`. Šajā gadījumā simboli >> aizver divas atvērtās grupas pēc kārtas. Lai tas tiešām būtu atpazīts, ka grupu aizvēršana, lekserim jāzina simbolu konteksts, vai arī jāseko valodas gramatikai, jo parasti simbolu kombinācija >> nozīmē operāciju pārbīdei pa labi.

2. Gadījumā, ja lietotājam ir dota iespēja definēt savus operatorus, ieviešot operatoru, kas pārklāj eksistējošos, ir jāmaina leksēšanas likumus. Piemēram, ja lietotājs definē unāru operatoru `+-`, tad izteiksmei `+-5` ir jābūt saprastai ka `(+-, 5)`, nevis ka `(+, -, 5)`.

Apskatītie piemēri parāda to, ka automātisku parsētāju ģeneratoru lietošana dažreiz var būt tik pat sarežģīta, ka parsētāja rakstīšana ar rokām. Parsētāju ģeneratori nav piemēroti pakāpeniskām valodas sintakses evolūcijām, to izveidoto parsētāju pārģenerēšana aizņem laiku un resursus.

Izrādās, ka daudzām eksistējošām valodām parsētāji tiek rakstīti manuāli[1]. Tas nozīmē, ka visticamāk arī kross-kompilatoru būs jāraksta manuāli, risinot eksistējošās gramatikas konfliktus, un oriģinālvalodas gramatikas izmaiņu gadījumā būs jāpastrādā abi kompilatori.

Šīs darbs izvirza pieeju, kas lietos eksistējošo valodas parsētāju par pamatu savam darbam un piedāvās likumu kopu, kas ļaus paplašināt valodas gramatiku. Taču patvaļīgas lietotāja iniciētas gramatikas izmaiņas (gramatikas likumu pievienošanas un dzēšanas) var novest pie nekontrolējamas valodas evolūcijas. Tāpēc aprakstāmā pieejā tiek piedāvātas ierobežotas izmaņu iespējas, kas tiks daļēji kontrolētas ar speciāli izveidotas tipu sistēmas palīdzību.

Projektētā 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. Transformācijas korektums šī darba ietvaros nozīmē to, ka transformācijas izejas daļiņu virkne būs korekta uzrādīta transformācijas beigu tipa vienība. Piemēram, ja makro tiek uzrādīts, ka beigu transformācijas tips būs kaut kāds identifikators, bet pēc pārveidošanas no daļiņām sanāks izteiksme, šāds makro netiks akceptēts.

Ļoti līdzīgu uzdevumu, izņemot tikai transformāciju korektuma pārbaudi, pilda arī vispārējie teksta priekšprocesori. Jebkura priekšprocesora viens no galveniem mērķiem ir vienas

[1] Piemēram C/C++/ObjectiveC kompilators GNU GCC [8], clang kompilators LLVM [18], JavaScript Google V8 [9]

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 atdalošo var definēt jebkādu simbolu. Tad apstrāde ir šo virkņu aizvietošana ar citām izveidotām virknēm.

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

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

Pirmkārt, šādam makro nav iespējams statiski izsecināt rezultāta tipu tā ielasīšanas brīdī un izveidot kļūdas paziņojumu gadījumā, ja makro satur transformācijas, kas varētu sabojāt pārveidojamo kodu. Transformācijas rezultāta izrēķināšana ir iespējama tikai pārējā programmas teksta apstrādes laikā, 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)` tiks pārveidots par `(5, 6) 7`.

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

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

Bet 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.

Pat neņemot vērā grūtības mēģinot pierādīt transformācijas korektumu, teksta 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 daļiņu un reducēto produkciju – pseido-daļiņu virknes.

Lai padarītu transformācijas sistēmu vairāk spēcīgu un ļautu atpazīt vispārīgākas virknes, tās makro šabloni tiks paplašināti ar regulāro izteiksmju elementiem. Šādi šabloni ļaus meklēt plašākas sakritības joprojām saglabājot iespēju kontrolēt pārveidojumu 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.

1.3. Sistēmas koncepcija

Aprakstāmā transformācijas 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 kaut kādas konstrukcijas ar citām noteikti definētam konstrukcijām. Otrā pieeja ir adaptīvās gramatikas - gramatikas, kas ļauj programmas kodam modificēt savas valodas gramatiku. Abas pieejas dod ļoti spēcīgus rīkus programmēšanas valodu izstrādē. Tomēr abām šīm pieejām ir savas problēmas un trūkumi, no kuriem šīs sistēmas projektēšanā mēģināja izvairīties.

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 vienīgi tekstu, tā nebūtu spējīga pārveidot šādu konstrukciju. Tiks apstrādāti pirmie divi `|`simboli, no `| (a |`, izveidojot nekorektu konstrukciju `abs((a) b) + c) |`. Šīs problēmas izvēlētais risinājums ir aprakstīts 1.3.1. apakšnodaļā.

Otrā problēma ir neiespējamība vispārīgā gadījumā kontrolēt dinamiskas gramatikas izmaiņas. Dinamiskas gramatikas ir ļoti spēcīgs rīks, kas mūsdienās gandrīz netiek lietots 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 1.3.2. apakšnodaļā.

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 bāzējoties uz vienu no plaši lietojamam parsētāju arhitektūrām. Kā tas tiks darīts ir aprakstīts 1.3.3. apakšnodaļā.

1.3.1. Teksta pārveidošana

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 abstraktos sintaktiskos kokus (AST), ko tas ir uzbūvējis. Šāda tipa priekšprocesori netiks apskatīti šajā darbā, jo to darbībai ir nepieciešams korekts sintakses koks. Aprakstāmās sistēmas gadījumā sintakses koka uzbūvēšana var būt neiespējama tad, kad gramatikā tika ieviestas kaut kādas jaunas konstrukcijas. Sistēmai jānostrādā pirms sintaktiskā koka izveidošanas. Otra no priekšprocesēšanas pieejām ir leksiskā. Leksiskie priekšprocesori tiek palaisti pirms pirmkoda parsēšanas un tiem 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, tātad nozīmē ka būs divreiz jāimplementē sintakses atpazīšana.

Lai izvairīties no šīs problēmas tika izvēlēts apstrādāt nevis programmas tekstu, bet gan programmas daļiņu un reducēto produkciju – 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 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, kas tiek aizvietota gramatikas likuma reducēšanas brīdī. Patiesībā tā ir atomāra vienība.

Tā kā aprakstāmā sistēma tiek izstrādāta tā, lai tā varētu darboties zinot pēc iespējas mazāk 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, kas aizvietos atrasto. 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 uzskatīta par parastu tekstu un var tikt pārrakstīta. Loģiskāk būtu, ja konkrētā tvērumā definēti makro tiktu mantoti līdzīgi ka mainīgie, kas nozīmē, ka šabloni, kas ir specifiski tvērumam, būtu ar lielāku prioritāti ka tie, kas definēti vispārīgākā tvērumā.

Sistēmas sakrišanas meklēšanas mehānisms tiks izstrādāts ņemot vērā programmas tvēru-

[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.

ma maiņu. Tātad šabloni, kuri tiek ieviesti konkrētā tvērumā, strādās tikai tā ietvaros. Pēc izejas no tvēruma ielasītie makro tiek aizmirsti un tālāk sistēma strādā bez tiem.

1.3.2. Dinamiskas gramatikas

Ir dažādas iespējas lietot dinamiskas gramatikas. Viena no tām ir pieeja, kas ļauj kontrolēt tā saukto programmas statisko semantiku. Par statisko semantiku dažādos rakstos sauc kontekst-atkarīgu sintaksi, piemēram, attiecību starp mainīgā deklarāciju un tā lietošanas ierobežojumiem. Otra pieeja ļauj pa tiešo pievienot un dzēst valodas gramatikas likumus. Tā kā aprakstāmā sistēma nav piemērota statiskai tipu pārbaudei, bet gan tiek veidota lai varētu paplašināt eksistējošo sintaksi, tā pēc būtības ir tuvāka otrai dinamisko gramatiku lietošanas pieejai.

Transformācijas sistēma neļaus pievienot un dzēst apstrādātās valodas gramatikas likumus. Toties tā ļaus pievienot jaunas konstrukcijas pievienojot likumus makro sistēmas kopai. Katrs makro 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.

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 2.3.3. apakšnodaļā.

1.3.3. 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 2.1. apakšnodaļā.

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

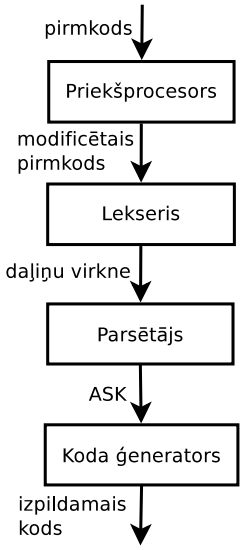
1.4. Sistēmas vieta kompilēšanas procesā

Šī nodaļa vienkāršoti apskata kā parasti notiek programmu kompilēšanas process un iezīmē procesā vietu priekš transformācijas sistēmas.

Parasti kompilēšanas process sastāv no diviem posmiem, no analīzes un no sintēzes. Analīzes posmā programma tiek sadalīta sastāvdaļās un no šīm daļām tiek veidota programmas struktūra. Šī struktūra tiek lietota lai izveidot starpfāzes reprezentāciju programmai - abstrakto sintakses koku (ASK). Analīzes fāzē var tikt atrastas sintaktiskas vai semantiskas izejas koda kļūdas un kompilēšanas process var tikt pārtraukts. Sintēzes posmā, savukārt, no ASK un papildus informācijas par programmas datiem, kompilators veido izpildāmo kodu.

Šeit netiks apskatīts izpildāmā koda veidošanas process, jo aprakstāmā sistēma iekļaujas parsēšanas gaitā. Tomēr vajag atzīmēt, ka koda ģeneratoram ir nepieciešams korekts sintakses koks, tātad tam ir ļoti stingri definēta saskarne. Visām izejas koda pārveidošanām, jā vien tās tiek ieviestas, ir jāstrādā tā, lai analīzes fāzē izveidotais ASK būtu korekts koda ģeneratora ieejas koks.

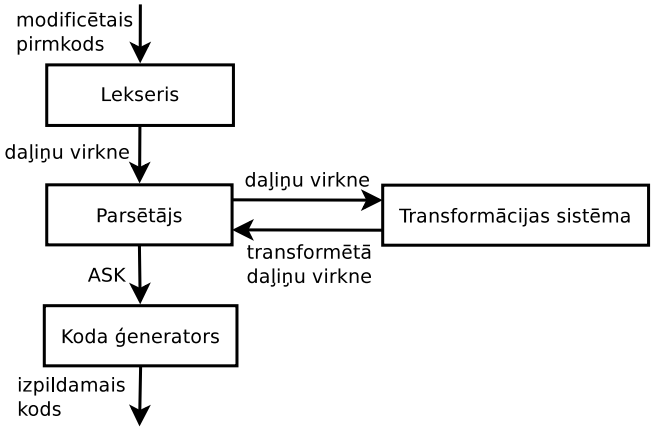
Parasti analīzes process notiek sekojoši (sk. 1.1. attēlu). Izejas programmas teksts tiek apstrādāts ar priekšprocesoru, kas aizvieto vienus izejas teksta gabalus ar citiem, kas tika tam definēti. Tad pārveidotais kods tiek apstrādāts ar lekseru, kas sadala programmas tekstu jēdzīgās sastāvdaļās - daļiņās. Tālāk parsētājs transformē daļiņu virkni atbilstoši valodas gramatikai izveidojot ASK.



1.1. att. **Kompilācijas process**

Transformācijas sistēma tiek veidota tā, lai tā apstrādātu daļiņu virknes paralēli ar parsētāju (sk. 1.2. attēlu). Tās funkcijas tiks izsauktas pirms vai pēc parsētāja produkcijas reducēšanas. Ja transformācija tiks izsaukta pirms parsētāja darbības, tad transformācijas likumiem lielāka prioritāte nekā standartiem produkcijas reducēšanas likumiem, piem. būs iespēja pārdefinēt izteiksmes `a+b` uzvedību ar transformācijas palīdzību. Ja transformācija tiks izsaukta pēc parsētāja darbības, tas nozīmēs, ka nekāds standarta parsēšanas likums nav piemērots dotās virknes apstrādei un ir nepieciešamas modifikācijas. Transformācijas sistēmas uzvedības nav atkarīga no

izvēlētās pieejas.



1.2. att. **Kompilācijas process ar iekļauto transformācijas sistēmu**

Zemāk ir parādīti makro ielasīšanas soļi, kas notiek, kad parsētājs sastopas ar makro ierakstu programmas pirmkodā:

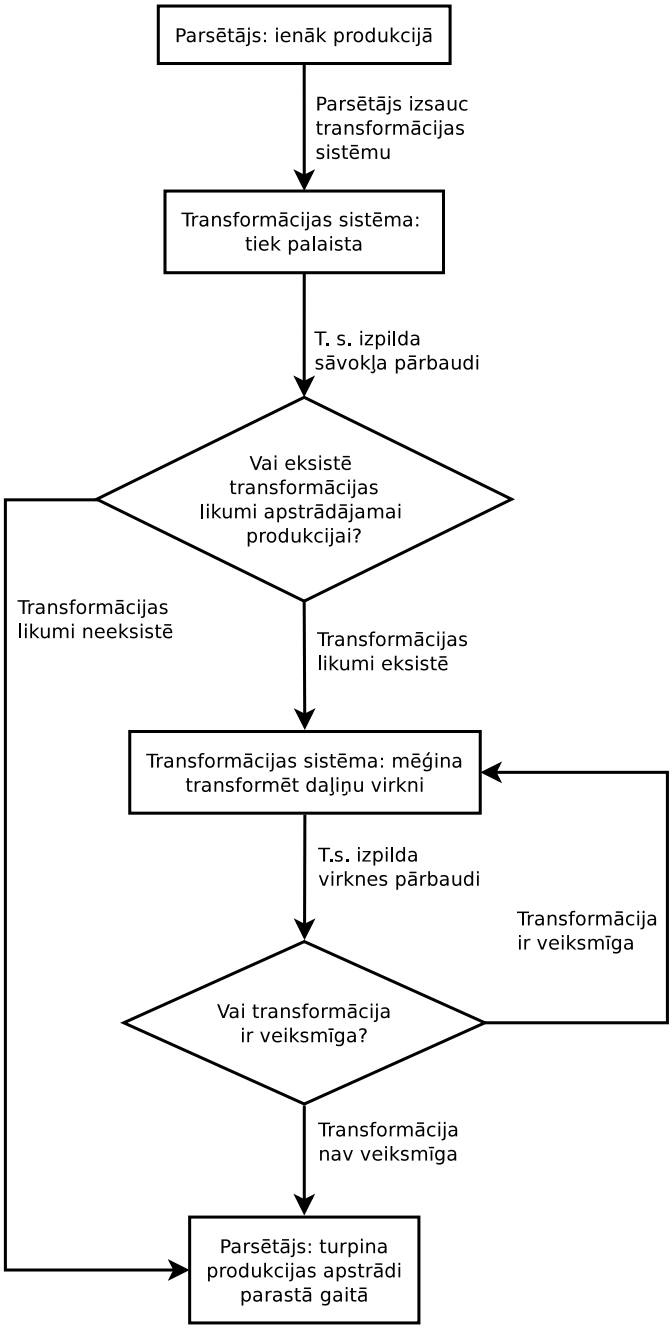
1. Parsētājs atpazīst makro sākšanos un izsauc transformācijas sistēmu.
2. Transformācijas sistēma ielasa makro.
3. Tipu apakšsistēma pārbauda makro tipu korektumu. Ja tipi ir korekti, turpina uz 4. soli. Ja ielasītā makro tipi nav korekti, turpina uz 5. soli.
4. Transformācijas sistēma saglabā makro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

Makro atpazīšanas soļi, gadījumā, jā transformācijas sistēma tiek izsaukta pirms parsētāja mēģinājuma apstrādāt produkciju:

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 2a. soli. Citādi tā nedara neko un parsētājs turpina darbu ar 3. soli.
   (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 2b. soli. Ja šādas sekvences nav, turpina uz 3. soli.
   (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 2a. soļa.

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

1.3. attēls parāda transformācijas sistēmas palaišanu un darbību virkni diagrammas veidā.



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

2. 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 valodas pašmodificēšanos lietojot konstrukciju aizvietošanu, kas vienlaikus nodrošinās valodas sintakses paplašināšanu un sākotnējās valodas gramatikas nemainīgumu. Tajā pašā laikā sistēma būs samērā stabila pret kļūdām dēļ tā, ka tā strādās tikai konkrētās gramatikas produkcijas ietvaros un tā, ka tā pārbaudīs tipus jaunizveidotām virknēm[1].

Sistēmas raksturīga īpašība ir tas, ka tā nav piesaistīta konkrētai programmēšanas valodai. Tā varēs tikt pielāgota un integrēta dažādu valodu parsētājos, kuru arhitektūra atbilst dažiem nosacījumiem. Tai nav ierobežojumu pret atbalstāmo sintaksi, jo tā strādās ārpus valodas gramatikas un tai būs nepieciešamas minimālas zināšanas par tās sintaksi un semantiku, kas tiks iegūtas lietojot parsētāja saskarni.

2.1. 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.

Daļiņu virkne Parsētājam jāprot aplūkot daļiņu virkni ka abpusēji saistītu sarakstu, lai eksistētu iespēja to apstaigāt abos virzienos. Tam arī jādod iespēju aizvietot kaut kādu daļiņu virkni ar jaunu un ļaut uzsākt apstrādi no patvaļīgas vietas daļiņu virknē.

Tas ir nepieciešams, lai transformācijas sistēma varētu aizvietot transformētās virknes visā programmas daļiņu virknē.

Pseido-daļiņas Parsētāji parasti pielieto (reducē) gramatikas likumus ielasot daļiņas no ieejas virknes. Pseido-daļiņa, savukārt, ir produkcijas redukcijas rezultāts. Tas tiek attēlots ka atomārs ieejas plūsmas elements, bet īstenībā attēlo kaut kādu valodas gramatikas netermi-

[1] Ir saprotams, ka sistēmu, kas pavisam nepielaiž kļūdu parādīšanos ir gandrīz neiespējami izveidot.

nāļi. Viena no pseido-daļiņām, piemēram, ir izteiksmes daļiņa - `{expr}`, kas var sastāvēt no daudziem dažādiem daļinām (piem. `(a+b*c)+d`).

Šāda tipa daļiņas ir nepieciešamas, lai šablonu sistēma varētu meklēt sakritības ar gramatikas produkcijām, neimplementējot gramatikas atpazīšanu. Piemēram, ja ir nepieciešams aizvietot visas izteiksmes pēc vienādojuma zīmes ar tām pašām virknēm, bet ar iekavām, tad varēs lietot izteiksmes pseido-daļiņu, nemēģinot šablonā aprakstīt visas iespējamas izteiksmes variācijas.

Vadīšanas funkcijas Sistēmas darbam ir prasīts, lai katra gramatikas produkcija tiktu reprezentēta ar vadīšanas funkciju (*handle-function*). Vadīšanas funkcijām jāeksistē katrai gramatikas produkcijai, un katra šāda tipa funkcija prot atpazīt un reducēt produkciju.

Ir svarīgi atzīmēt, ka šim funkcijām būs blakus efekti - daļiņu virknes pārveidošana pēc likuma reducēšanas, 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.

Ir-funkcijas Katra no vadīšanas funkcijām nāk pārī ar predikāta tipa funkciju (*is-function*). Šīs funkcijas pārbauda, vai tajā vietā daļiņu virknē, uz kura dotajā brīdī atrodas parsētāja radītājs, ir atrodama apakšvirkne, kas atbilst dotās produkcijas aprakstam. Šādu funkciju pielietošana nemaina parsētāja stāvokli.

Sakrišanas funkcijas Katras vadīšanas funkcijas darbības sākumā vai beigās (tas ir atkarīgs tikai no implementācijas izvēles) tiek izsaukta tā sauktā sakrišanas funkcija (*match-function*). Sakrišanas funkcija ir transformācijas sistēmas saskarne, kas, zinot, kādā produkcijā uz doto brīdi strādā parsētājs, mēģina izdarīt daļiņu virknes transformāciju.

Tā pārbauda, vai tā vieta daliņu 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 tas var tikt veiksmīgi savienotas ar aprakstāmo transformācijas sistēmu un ļaut programmētājam ieviest modifikācijas oriģinālās valodas sintaksē.

2.2. Makro sistēmas sintakse

Makro sintakse ir veidota transformācijas likuma izskatā. Transformāciju likums sastāv no divām daļām. Kreisā puse satur šablonu un produkciju, kurā tā var tikt pielietota. Makro šablons satur regulāro izteiksmi no daļiņām un pseido-daļiņām, kas tālāk tiek izmantota lai atrast virkni, kurai šī transformācija ir pielietojama. Labā makro puse satur transformācijas funkciju un produkciju, kurai ir jāatbilst transformācijas rezultātam.

Makro izteiksmes strādā stingri kaut kādas produkcijas ietvaros, tāpēc makro šablona sintaksē tiek lietoti produkciju nosaukumu apzīmējumi. Tie tiks lietoti lai pārbaudītu pseido-daļiņu virknes korektumu sākotnējās gramatikas ietvaros pēc sintakses izmaiņu ieviešanas. Par tipu pārbaudi tiks runāts zemāk šajā nodaļā.

Apskatīsim *match* funkciju likumus, kas modificē apstrādājamās gramatikas produkcijas uzvedību. *Match* makro sintakses vispārīgo formu var redzēt 2.4. attēlā.

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

2.4. att. ***Match*** **makro sintakses vispārīgā forma**

Šīs apraksts ir uztverams sekojoši. Ja produkcijas `prod1` sākumā ir atrodama pseido-daļiņu virkne, kas atbilst regulārai izteiksmei `regexp`, tad tai tiek nosacīti piekārtots vārds `v`. Mainīgais `v` ir visu atrasto daļiņu virkne, kas var tikt lietota makro labajā pusē pārveidošanas funkcijas izpildē. Tātad ja tāda virkne `v` eksistē, tā tiek aizstāta ar pseido-daļiņu virkni, ko atgriezīs `f(v)` un kurai ir jāatbilst gramatikas produkcijas `prod2` likumiem.

Regulārā izteiksme `regexp` ir vienkārša standarta regulārā izteiksme, kuras gramatika ir definēta 2.5. attēlā. Pagaidām sistēmas prototipa izstrādē tiek lietota šāda minimāla sintakse, bet nepieciešamības gadījumā tā var tikt paplašināta.

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

2.5. att. **Regulāro izteiksmju gramatika uz pseido-daļiņām**

Piemēros uzskatāmības nolūkos `v` virkne būs uzskatīta par apstaigājamu un lietota ar indeksiem, kas norāda atrastās virknes elementa numuru.

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 2.6. attēlā izdarītu šo substitūciju, ļaujot programmētājam lietot ērtāku funkcijas pierakstu.

```
match [{expr}] v = {|} {expr} {|}
    → [{expr}] {id:abs} {(} v[1] {)}
```

2.6. att. **Makro piemērs #1**

Šajā piemērā šablons `{|} {expr} {|}`, strādājot `{expr}` produkcijas ietvaros, atradīs sakritību ar trim daļiņām, `|`, kaut kāda izteiksme, `|`. Transformējot atrasto elementu sarakstu tiek veidots jauns saraksts no `{id:abs}`, `{(}`, atrastās izteiksmes `v[1]` un `{)}`. Izveidotam sarakstam jāatbilst produkcijai `{expr}`.

Vēl viens korektā makro piemērs: pieņemsim, ka funkcija `replace` ir definēta transformāciju valodā ar trim argumentiem, un darba gaitā tā jebkurā pseido-daļiņu virknē, kas padota ka pirmais arguments, aizvieto elementus, kas sakrīt ar otro argumentu, ar trešo funkcijas argumentu. Pieņemsim, ka mums ir nepieciešams aizvietot visas funkcijas `sum` izsaukšanas reizes ar tās argumentu summu. Funkcija `sum` ir iespējams padot patvaļīgu argumentu skaitu, kas ir lielāks par vienu. Šādā gadījumā makro, kas parādīts 2.7. attēlā, izpildīs nepieciešamu darbību.

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

2.7. att. **Makro piemērs #2**

2.3. Transformācijas sistēmas apakšsistēmas

Šī nodaļa parāda sistēmas sadalīšanu uz trim neatkarīgam daļām. Pirmā no tām ir sakrišanu meklēšanas daļa. Tā daļiņu virknē atrod makro šablonu satikšanas reizes. Otrā ir atrasto virkņu apstrādes daļa. Tā pārveido sakrišanas mehānisma atrasto daļiņu 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 2.3.1. apakšnodaļā. Sīkāk šīs apakšsistēmas īpašības un tās prototipa realizācija ir aprakstīta 3. nodaļā.

Ir nepieciešams izveidot mehānismu, kas ļaus transformēt makro kreisās puses akceptētu pseido-daļiņu virkni, izveidojot virkni, kas to aizvietos. Lai to izdarītu ir nepieciešama kaut kāds papildus rīks, par kuru ies runa 2.3.2. apakšnodaļā.

Ir plānots, ka transformāciju sistēma varēs atpazīt nepareizi sastādītus makro šablonus lietojot tipu kontroli. Šīs pieejas bāzes principi ir aprakstīti 2.3.3. apakšnodaļā. Jāņem vērā tas, ka lai šī sistēma varētu tikt pielietota, izvēlētai transformāciju valodai jāpiemīt tipu secināšanas (*type inference*) īpašībai.

2.3.1. Sakrišanu meklēšana

Šablonu apstrādei ir nepieciešama kāds sakrišanu meklēšanas mehānisms. Tā kā šabloniem tika izvēlēts lietot regulāro izteiksmju elementus, parastā virkņu saildzināšana nedos vēlamo rezultātu. Tāpēc tika izvēlēts veidot apakšsistēmu, kas varēs apstrādāt un saglabāt ielasītās regulārās izteiksmes un veikt ieejas virknes apstrādi, meklējot sakrišanas.

Galvenais princips uz kura jābāzējas šīs apakšsistēmas izstrādē minimāls apstrādes laiku. Tātad apakšsistēmas koncepcija tika izveidota tā, lai jebkurā laika momentā šablonu sakrišanu meklēšanai būtu nepieciešams lineārs laiks un tikai viena daļiņu 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, katrai daļiņai 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.

Sīkāk šīs apakšsistēmas prototipa īpašības un lietotie algoritmi ir izklāsti 3. nodaļā.

2.3.2. Daļiņu virkņu apstrāde

Gadījumā, ja šablonu sistēma nesaturētu regulāro izteiksmju sintaksi (it īpaši $*$), būtu iespējams transformēt daļiņu virknes ar pašu makro palīdzību. Atrastās daļiņu 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 daļiņu 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, ku-

ra 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. Apstrādes sistēmas mērķis ir ļaut izveidot atrasto daļiņu permutācijas ar kādiem papildinājumiem nepieciešamības gadījumā.

Galvenais šīs valodas pielietojums ir dot iespēju apstaigāt pseido-daļiņu 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 veidot skaitītājus un rekursijas izejas nosacījumus.

Tiek plānots, ka šī valoda arī ļaus nepieciešamības gadījumā izpildīt daļēju novērtējumu izteiksmēm. Tas nozīmē, ka valodai jāsatur saskarne, kas ļaus piekļūt pie daļiņas vērtības. Šim mērķim ir domāta funkcija `value`, kas ir pielietojama pseido-daļiņām ar skaitlisku vērtību, piemēram, lai dabūt skaitli `5` no pseido-daļiņas `{int:5}`. Valoda arī ļaus izveidot jaunas daļiņas ar izrēķinātu vērtību.

Funkcija `type`, savukārt, ļaus pārbaudīt daļiņu 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.

2.3.3. Tipu sistēma

Šī nodaļa tikai vispārīgi apraksta tipu secināšanas sistēmas ideju, tā kā uz doto brīdi tā atrodas izpētes stadijā. Tipu secināšanas ideja ir bāzēta uz Bodo Šulca rakstu par masīvu tipu izvadīšanas sistēmu [15].

Kā bija redzams 2.4. attēlā, katrā makro pusē ir atrodams produkcijas nosaukums, `[prod1]` un `[prod2]`. Tas tiek darīts tādēļ, lai, kad dotais makro ir ielasīts, varētu izsecināt 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 daļiņu virkni, ir atļauta ieejas virkne priekš produkcijas `prod2`.

Lai to paveikt, ir nepieciešams izsecināt funkcijas `f` no virknes `v` rezultāta tipu un pārbaudīt, vai tas der produkcijai `prod2`. `f(v)` rezultāta tips būs kaut kāda regulāra izteiksme, ko veidos makro kreisās puses regulārās izteiksmes elementi. To attiecību ar `prod2` var pārbaudīt dažos

veidos, piemēram, izveidojot no tās gramatiku un pārbaudot, vai tā ir apakšgramatika produkcijai `prod2`. Šī darba ietvaros tika izvēlēts veidot regulārās izteiksmes no gramatikas produkcijām un pārbaudīt, vai produkcijas izteiksme iekļauj sevī funkcijas rezultāta izteiksmi.

Šajā darbā netiks apskatīts jautājums, kādā veidā tiks izveidota regulārā izteiksme priekš katras gramatikas produkcijas. To varētu izveidot programmētājs vai 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 $r_1 \sqsubseteq r_2 \Rightarrow$ seko $min(det(r_1)) \sqsubseteq min(det(r_2))$. 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-daļiņu 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-daļiņu 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 virknes garumu intervālu un izvadīt brīdinājumus gadījumā, ja ir iespējama funkcijas $head(x)$ izsaukšana no tukšā saraksta. Visām pārējām iebūvētām funkcijām – $cons$, $tail$ – ir iespējams analoģiski izsecināt rezultātu tipus. Tā kā visas darbības saraksta apstrādē tiek izpildītas

[1] Sk. 3.4. nodaļu.

ar iebūvēto funkciju un rekursijas palīdzību, arī to rezultātiem ir iespējams izrēķināt vispārīgus tipus.

3. 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 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.

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

3.1. Atļautā makro sintakse

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

Prototips arī ļauj veidot regulārās izteiksmes ar specificētu daļiņu vai pseido-daļiņu 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.

Iespējamās sintakses piemērs: `{id:aaa} {(} {real} ({,} {int})* {)}`. Šāda izteiksme sagaidīs funkciju ar nosaukumu `aaa`, vismaz vienu `real` tipa parametru un ne vienu vai vairākus `int` tipa parametrus. Izteiksme `{id} {(} ( {real}` int))| sagaidīs jebkādu funkciju ar vienu parametru, kas var būt vai nu `int` tipa, vai nu `real`.

3.2. Vispārīgā pieeja

Prototips imitē darbu reālajā vidē, saņemot daļiņas no ieejas plūsmas pa vienam no atsevišķas saskarnes. Daļiņu plūsmas saskarne imitē parsētāja saskarni. Kamēr prototips nav saņēmis nevienu regulāro izteiksmi, tas ignorē daļiņu plūsmas apstrādes izsaukumus. Tiklīdz prototipam atnāk izsaukums apstrādāt daļiņu 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 daļiņu apstrādes pieprasījums, sistēma izpilda pārejas starp automātu stāvokļiem, meklējot sakrišanas, un atceras daļiņas, kuras jau ir nolasījusi. Sistēma atrod garāko virkni, kas atbilst kādam no šabloniem un tad atgriež tās identifikatoru un nolasīto daļiņu 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 daļiņu 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 daļiņu virkne tiks apstrādāta ar standartiem valodas likumiem.

3.3. 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.

Apskatot 2 makro raksturīgos parametrus - prioritātes pa tvērumiem un sakrītošo virkņu garumus, var rasties 4 konfliktu varianti. No tiem tiks apskatīti 3 konfliktu veidi, jo garumu konflikts tiks risināts vienādi gan makro no diviem tvērumiem, gan makro vienā tvērumā.. 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 atpazīst vienu un to pašu virkni ka jau eksistējošs šablons no vispārīgāka tvēruma. Trešais var rasties tad, kad divas izteiksmes akceptē virknes ar dažādiem garumiem.

3.3.1. 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.

3.3.2. 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.

3.3.3. 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 daļiņu virkne `{int:4} {,} {int:6} {,}` tiks akceptēta ar otru šablonu, neskatoties uz to, ka augstākās

prioritātes šablona sakritība tika konstatēta agrāk.

3.4. 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 daļiņu. Regulāro izteiksmju apstrādes gaitā daļiņas `{id:foo}` un `{id}` tiek uzskatīti par dažādiem, kaut arī `{id:foo}` ir apakšgadījums daļiņai `{id}`. Šīs fakts tiek iegaumēts tikai sakrišanu meklēšanas gaitā.

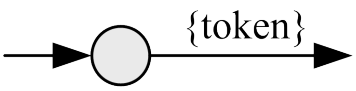
Algoritmi, kas tika lietoti prototipa implementācijā tika ņemti no dažādiem literatūras avotiem, kas ir norādīti apakšnodaļās, un adaptēti lietošanai aprakstāmos nolūkos.

3.4.1. Regulāro izteiksmju pārveidošana NGA

Regulāro izteiksmju translēšana uz nedeterminētu galīgu automātu ir diezgan vienkārša[1]. 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.

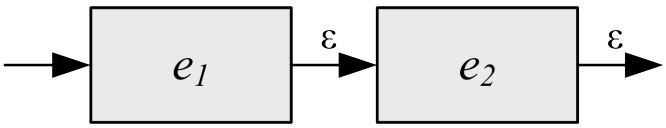
Nedeterminēts galīgs automāts (NGA) veselai regulārai izteiksmei ir izveidots to daļējiem automātiem katrai regulārās izteiksmes daļai. Katram operatoram tiek piekārtota attiecīga konstrukcija. Daļējiem automātiem nav akceptējošu stāvokļu, tiem ir pārejas uz nekurieni, kuras vēlāk tiks lietotas lai savienotu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

3.8. attēlā ir parādīts NGA vienai daļiņai $token$.



3.8. att. **Automāts vienai daļiņai**

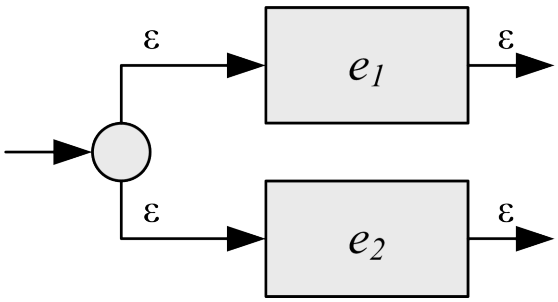
3.9. attēlā ir parādīts NGA divu automātu konkatenācijai $e_1 e_2$.



3.9. att. **Automāts divu automātu konkatenācijai**

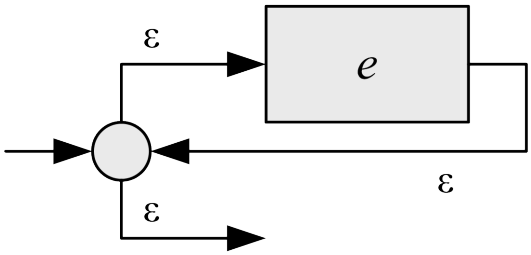
3.10. attēlā ir parādīts NGA izvēlei starp diviem automātiem $e_1|e_2$.

[1] Pieradījumu tam, ka katra regulāro izteiksmju definēta valoda ir atpazīstama ar galīgiem automātiem sk. [13]



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

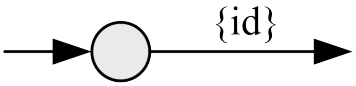
3.11. attēlā ir parādīts NGA priekš konstrukcijas e_1*.



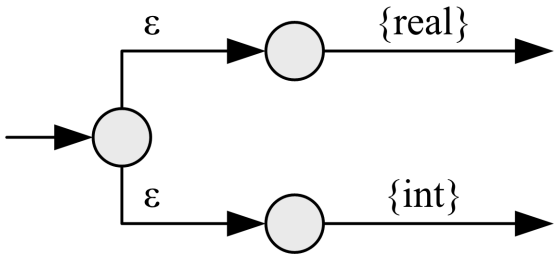
3.11. att. **Automāts automātu virknei**

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

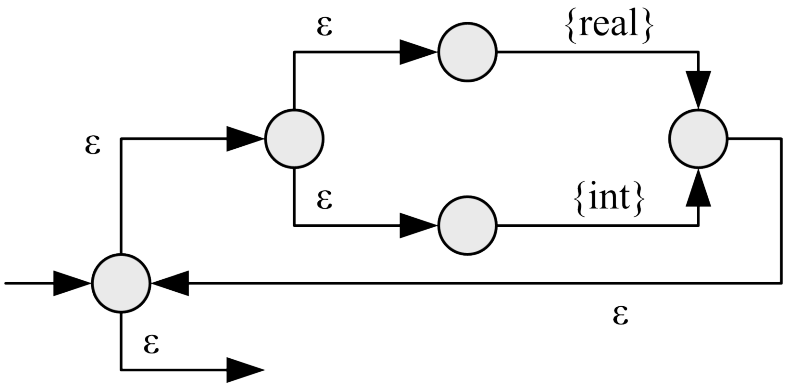
Apskatīsim piemēru - izteiksmi `{id} ({real} | {int}) *`. Sākumā tiek izveidoti NGA izteiksmes daļām. 3.12. attēls parāda NGA priekš `{id}`. 3.13. attēls parāda NGA priekš daļas `{real} | {int}`. 3.14. attēls parāda NGA priekš `({real} | {int}) *`.



3.12. att. **Automāts daļiņai `{id}`**

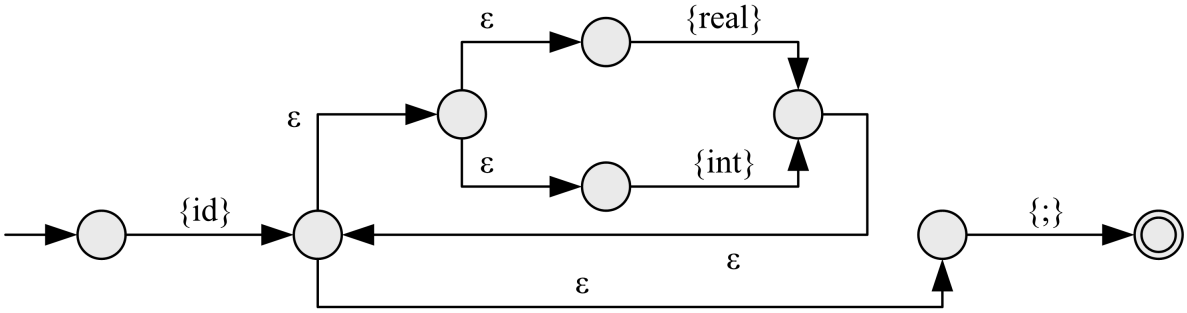


3.13. att. **Automāts izteiksmei `{real} | {int}`**



3.14. att. **Automāts izteiksmei `({real} | {int}) *`**

Tad izveidotie automāti var tikt savienoti un beigās tiem tiek pievienots akceptējošais stāvoklis (3.15. attēls).



3.15. att. **Automāts izteiksmei `{id} ({real} | {int}) * {;}`**

Tā tiek izveidots nedeterminēts automāts katrai regulārai izteiksmei. [7], [1]

3.4.2. NGA determinizēšana

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. [11], teorēma 2.11.

Algoritms ir paņemts no [1].

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.

3.1. tabulā 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.

3.1. 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 $\varepsilon - closure(move(T, a))$. 3.16. attēls 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.

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 path 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

3.16. att. **Automāta determinēšanas algoritms**

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 $O(n * m)$. [1, 11]

3.4.3. DGA 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 neakceptējošs vienlaikus). Divi neatšķirami stāvokļi ir ekvivalenti[3].

Algoritms ir paņemts no [1].

Algoritms 2: DGA minimizēšana

Ieeja: DGA D.

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

[1] Šī fakta pierādījumu sk. [2]

[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] [11] nodaļa 4.4

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.

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.

3.17. attēls 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
$\Pi_{final} = \Pi$

3.17. 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 izveidotas 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 sliktākajā gadījumā ir $O(n^2)$, kur n ir sākotnējā automāta stāvokļu daudzums. Tomēr vidēji algoritma darbības sarežģītība ir $O(n \log n)$. [2]

3.4.4. DGA 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 determinēšanas algoritma adaptācija. 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.

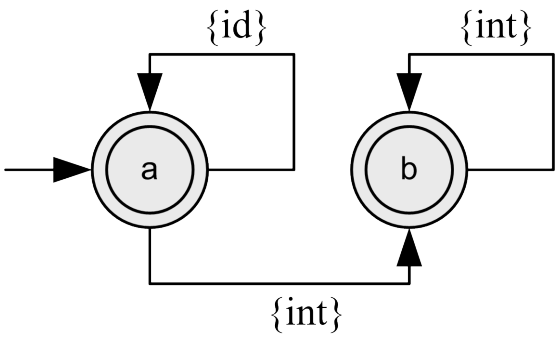
Algoritms 3: Divu DGA apvienošana

Ieeja: DGA D_1 un D_2.

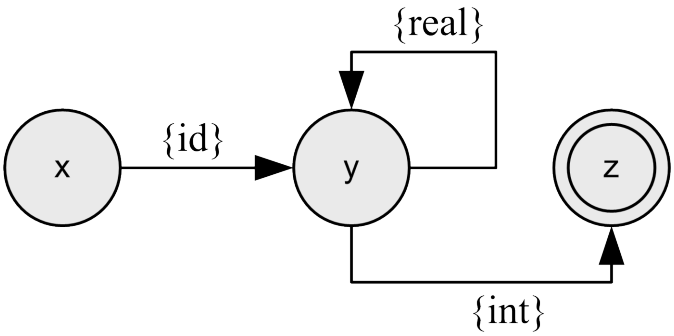
Izeja: Jauns DGA D', kas apvieno D_1 un D_2.

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. 3.18. attēls parāda determinētu minimālu automātu priekš regulārās izteiksmes `{id}* {int}*`. Tas satur divus stāvokļus, `a` un `b`, kuri abi ir akceptējoši. 3.19. attēls, savukārt, parāda DGA priekš izteiksmes `{id} {real}* {int}`.

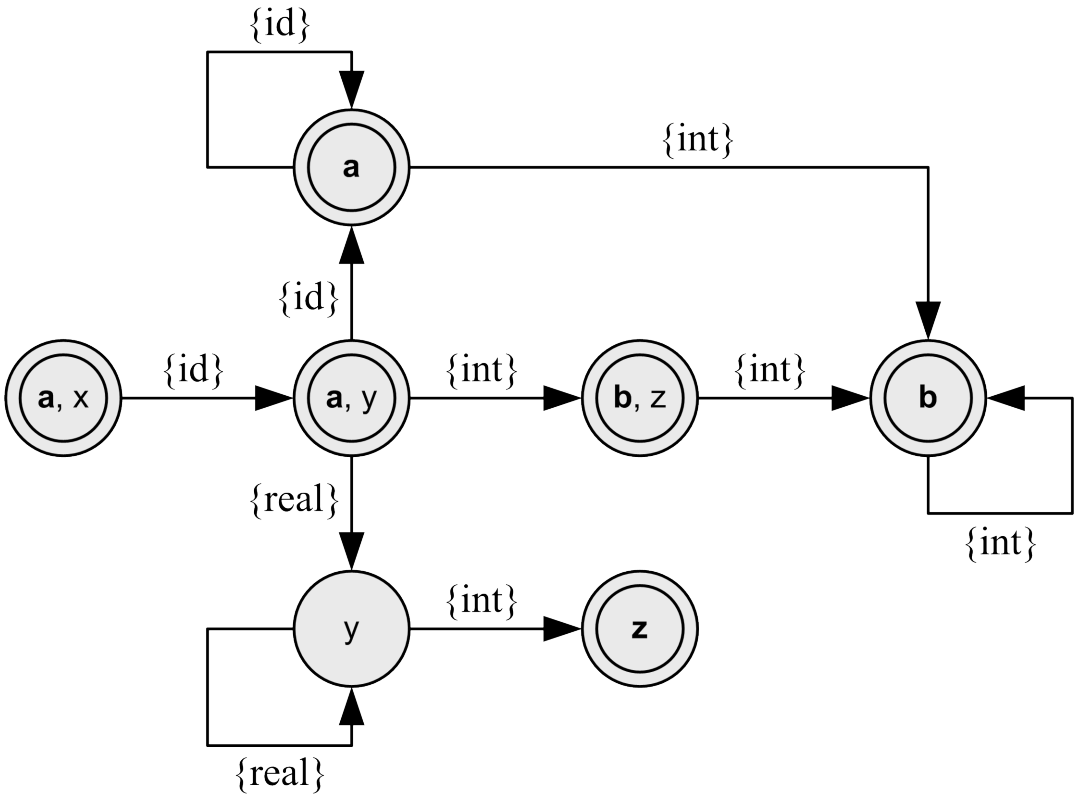


3.18. att. **Automāts izteiksmei `{id}* {int}*`**



3.19. att. **Automāts izteiksmei `{id} {real}* {int}`**

To apvienotais automāts ir parādīts 3.20. attēlā.



3.20. att. **Automāts izteiksmju `{id}* {int}*` un `{id} {real}* {int}` apvienojumam**

3.21. attēls 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 daļiņu 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 būt tikai viens stāvoklis, gadījumā, ja no kāda r_i un t_i neeksistē pāreja pa doto daļiņu.

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

3.21. att. **Divu automātu apvienošanas algoritms**

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.

3.4.5. Tvērumu apstrāde

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.

3.4.6. 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 pārbau-

dāmās virknes garums.

Diemžēl pilnībā lineāra laika sakrišanu meklēšana nav iespējama tādēļ, ka prototips dod iespēju lietot makro ar daļiņu vērtībām. Piemēram, eksistē 2 makro, viens no kuriem gaida daļiņu `{id}`, un otrs `{id:foo}`. Gadījumā, kad sakrišanu meklēšanas procesā parādās daļiņa `{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 nenoteiktību, tā var parādīties tikai augstāk minētā gadījumā un izveidot ne vairāk ka 2 ceļus vienlaikus. Tātad kaut arī nenoteiktība pastāv, tai ir ļoti maza iespējamība un maza ietekme uz sakrišanu meklēšanas laiku.

3.5. Izņēmumi

3.5.1. 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.

3.5.2. Daļiņu klašu mantošana

Sistēmai ir minimāla informācijas par valodas gramatiku, tai ir zināšanas tikai par daļiņu un produkciju nosaukumiem. Tieši tāpēc daļiņa `{real}` netiks uztverta ka `{expr}`, kaut arī racionāls skaitlis ir izteiksme. Tā kā sistēmai jābūt pēc iespējas vairāk neatkarīgai no valodas gramatikas, lai būtu universālākai, šī hierarhija nav iekodējama transformāciju sistēmā. To ir jānodrošina parsētājam, attiecīgi apstrādājot daļiņas apkopojot to nozīmi un veidojot saskarni priekš šablonu sistēmas.

3.5.3. Regulārās izteiksmes daļu grupēšana

Tādēļ, ka aprakstītā pieeja mēģina pēc iespējas minimizēt meklēšanas laiku, tā neatļauj veidot atrasto daļiņu grupēšanu, kā tas ir parasti pieņemts regulārās izteiksmēs. Daļiņu grupēšana un atpakaļnorādes (*backreferences*) uz daļiņu 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 daļiņu grupas. Tomēr šāda pieeja neatļaus sapludināt dažus automātus vienā, jo sapludināšanas procedūra sabojās grupēšanu.

3.5.4. Sapludinātā automāta minimizēšana

Automātu minimizēšana ir diezgan darbietilpīga operācija, tāpēc tā tiek izpildīta tikai uz atsevišķiem automātiem. Apvienotais visu šablonu automāts var nebūt minimāls, jo dažādu minimālu automātu apvienošana negarantē šo faktu. Bet tā kā apvienota automāta stāvokļu daudzums var būt ļoti liels, minimizēšanas izpilde var būt neefektīva. Tā kā minimizēšana samazina tikai automāta aizņemto vietu, nevis apstaigāšanas laiku, to 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.

3.6. 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, kas varētu būt lietotas tālākā izstrādē ir aprakstītas [20].

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 ar stāvokļu skaitu robežu, nevis tikai vienu. Šāda pieejā arī ir aprakstīta avotā [20].

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

3.7. Prototipa testēšana

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

3.7.1. Stresa testēšana

Prototips izstrādes laikā tika testēts ar lieliem automātiski ģenerēto datu apjomiem. Tika ģenerētas patvaļīgas 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 daļiņu 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.

3.7.2. Sistēmas testēšana

Prototipam tika izveidoti apmēram 20 testi, kas pārbauda to darbību iespējamās situācijās. 3.2. tabula parāda konceptuālu testu sadalījumu pa grupām. Dažas grupas pārklājas, jo, piemēram, tvērumu pārbaudošie testi pārbauda arī korektas regulāro izteiksmju prioritātes.

3.2. tabula
Prototipa testu sadalījums pa grupām

Testa nosaukums	**Testa apraksts**	**Kas tiek pārbaudīts**
Testi prioritāšu pārbaudei		
Testi ar dažiem šabloniem vienā tvērumā	Testu gaitā tiek izveidotas dažādas regulāras izteiksmes, kuru aprakstītās valodas pārklājas. Tās tiek ievietotas vienā tvērumā pēc kārtas. Tad tiek pārbaudīts, ka daļiņu saraksts tiek akceptēts ar pareizu šablonu attiecībā pret to prioritātēm.	Vai tiek korekti apstrādātas šablonu prioritātes viena tvēruma ietvaros.
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 daļiņu virkne.	Vai tiek korekti apstrādātas šablonu prioritātes viena tvēruma ietvaros.
Testi daļiņu vērtībām		
Testi ar daļiņu vērtībām	Testu gaitā tiek izveidotas dažādas regulārās izteiksmes ar daļiņu vērtībām un bez tām. Tiem tiek padotas dažādas daļiņu virknes.	Vai tiek korekti apstrādātas šablonu prioritātes un vērtību sakrišanas.
Testi ar vairākiem pieejamiem stāvokļiem vienlaikus	Testu gaitā tiek izveidotas dažādas regulārās izteiksmes ar daļiņu vērtībām. Tiem tiek padotas daļiņu virknes ar šādām pašām vērtībām, lai izveidotu situācijas, kad ir pieejami daži stāvokļi vienlaikus.	Vai tiek korekti apstrādātas situācijas, kad parādās nenoteiktība.
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 daļiņu saraksti.	Vai tiek korekti apstrādāta tvēruma parādīšanās. Vai tiek korekti apstrādātas šablonu prioritātes starp tvērumiem.

3.2. tabula
Prototipa testu sadalījums pa grupām

Testa nosaukums	Testa apraksts	Kas tiek pārbaudīts
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 daļiņu 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 daļiņu 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.
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 daļiņu 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 daļiņu virknes kuras neder nevienam no eksistējošiem šabloniem.	Vai sistēma korekti apstrādā situāciju, kad neviena sakrišana nav atrasta.

3.8. 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 daļiņu virkni meklējot sakrišanas

(`match_stream(stream)`). Integrēšanai prototipu būs jāpapildina ar iespēju padot produkcijas tipu daļiņu virkņu apstrādes funkcijām. Prototipa daļiņu saņemšanas funkcijas būs jāpārslēdz uz parsētāja piedāvāto saskarni daļiņu dabūšanai.

Prototipam ir nepieciešama vienkārša saskarne no eksistējošā parsētāja. Parsētājam jādot pieeju pie daļiņu virknes lasīšanas, kā arī jāprot aizvietot atrastās daļiņu virknes ar citām virknēm, iespējams, ar citu garumu. Parsētājam arī jāprot pārstartēt daļiņu virknes lasīšanu no aizvietotas virknes sākuma. Prototipam nepieciešamā saskarne ir implementēta Eq parsētājā.

Apvienojot parsētāja un sakritību meklēšanas prototipu būs nepieciešams ievietot prototipa funkciju izsaukumus katras produkcijas apstrādes sākumā. Gadījumos, kas parsētājs sastapās ar daļiņu, kas identificē makro sākšanos, būs nepieciešams izsaukt regulārās izteiksmes parsēšanas funkciju. Savukārt, kad tiek apstrādātas citas produkcijas, būs nepieciešams izsaukt sakrišanu meklēšanu. Abu funkciju izsaukumos būs nepieciešams padot arī produkcijas tipu, lai prototips varētu atšķirt, kādu no automātiem papildināt vai lietot sakrišanu atrašanai. Prototipa funkcijas būs jāizsauc ari tvērumu pārslēgšanu brīdī, lai tas varētu implementēt tvērumu makro prioritāšu sadalīšanu.

4. LĪDZĪGU DARBU APSKATS

Šī nodaļa apskata darbus, kuriem piemīt transformācijas sistēmai līdzīga funkcionalitāte. Šie darbi ir līdzīgi pēc savas būtības un dažreiz piedāvā alternatīvas pieejas aprakstītam uzdevumam. Toties lielāka šādu risinājumu daļa ir ļoti cieši piesaistīta pie konkrētas valodas, pat ne pie valodu klases. Zemāk tiks aprakstīti daži līdzīgi darbi un parādītas atšķirības starp tiem un aprakstīto pieeju.

Šīs darbs tika iedvesmots ar dažiem rakstiem par dinamisko gramatiku iespējām un pielietojumiem programmēšanas valodu izstrādē. 4.1. apakšnodaļa apskata izskatītus darbus par dinamiskām gramatikām.

Vispārīgi dinamiskas gramatikas un valodu dinamiska parsēšana gandrīz netiek lietota valodu implementēšanā. 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 4.2., 4.3., 4.4. un 4.5. apakšnodaļas 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.

4.1. Dinamiskas gramatikas

Ir dažas dinamisku gramatiku pieejas, kas, diemžēl, vairākumā ir tīri teorētiskas. Labu ieskatu adaptīvo gramatiku pieejās dod Heninga Kristiansena raksts par adaptīvām gramatikām –[6] un Džona Šutta maģistra darbs – [17]. Abi šie darbi apkopo visas uz to brīdi eksistējošās pieejas, bet, diemžēl, kopš šo rakstu laika citu ievērojamu variantu un implementāciju skaits ir ļoti mazs.

Dinamiskas gramatikas ir rīks, kas var ģenerēt neierobežotu kontekst-neatkarīgu gramatiku kopu. Katra gramatika ir veidota izejas teksta parsēšanas laikā sastopot speciālu konstrukciju. Dinamiska gramatika var tikt apskatīta ka parasto kontekstneatkarīgu gramatiku sekvence, ko definē programmas kods. Vispārīgi dinamiskas gramatikas var ļaut brīvi pievienot un dzēst savus produkciju likumus.

Toties pārsvarā dinamiskas gramatikas tiek lietotas lai pārbaudīt kontekst-atkarīgas sintakses korektumu, kas, piemēram, ir saistība starp mainīgā definīciju un tā lietošanu. Bez konteksta analīzes parsētājs nevarēs kļūdainu piešķiršanu, piemēram, šādā izteiksmē `int i; i = 'a'`. Šādas kļūdas parasti atrod kompilators pēc parsēšanas fāzes. Lietojot dinamiskās gramatikas katra mainīgā definīcija `type id;` varētu pievienot gramatikai likumu, ka `type` mainīgā vietā var būt mainīgais ar nosaukumu `id`.

Šī ideja izskatās ļoti pievilcīgi no pirmā skata, jo tā ļaus izlaist veselu pārbaužu kopu no kompilatora implementācijas. Diemžēl šādu gramatiku izveide nav tik eleganta, tā ir viegli izveidojama tikai tādiem vienkāršiem gadījumiem ka mainīgo un funkciju signatūru definīcija. Bloka beigu gadījumā gramatikai ir jāprot izmest ārā visus likumus, kas tika pievienoti bloka

iekšienē, kā arī kontrolēt mainīgo redzamību bloku ietvaros. Dinamisko gramatiku pieejas parasti arī neapstrādā rekursīvas deklarācijas. Vairāk informācijas var atrast Kristiansena rakstā [6].

Pjērs Bulliers savā rakstā par dinamiskām gramatikām [4] apskata iespēju lietot adaptīvas gramatikas valodas sintakses kontrolei. Tas apraksta, kā tās dod iespēju realizēt kontekstatkarīgas sintakses pārbaudi programmas parsēšanas, nevis kompilēšanas fāzē. Diemžēl, aprakstīta sistēma ir eksperimentāla un tikai prototipēta, nevis izveidota par lietojamu risinājumu. Tajā tiek lietota saskarne pie neeksistējoša orakula, kura mērķis ir risināt iespējamus gramatikas konfliktus.

4.2. 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.

Plašāka informācija par Lisp un tās makro sistēmu ir atrodama, starp citiem avotiem, grāmatā [16].

4.3. Forth

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

Forth satur tikai divus daļiņu tipus, skaitļus un visas citas valodas vienības - vārdus. Šāda pieeja ļauj rakstīt programmas dabiskā valodā, nelietojot iekavas lai padotu parametrus vārdiem-

funkcijām. Forth standarts definē speciālu vārdu kopu, kas ir iebūvēti valodā, bet tie arī var tikt pārdefinēti. Forth nesatur nekādus atslēgvārdus vispārpieņemtā nozīmē.

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 speciāli definētiem 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. Postfiksā 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.

Plašāka informācija par Forth valodu un tās iespējām ir atrodama tās mājaslapā [14].

4.4. Nemerle

Nemerle ir statiski tipizējama universāla programmēšanas valoda .NET platformai. Tai piemīt gan funkcionālas, gan objektorientētas, gan imperatīvās 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 ir statiski tipizējama un mēģina atbrīvot programmētāju no lieka darba lietojot tipu izvadīšanas iespēju un makro sistēmu. Tipu izvadīšana ļauj nerakstīt kodā tos tipus, kas var tikt izsecināti no koda gabala konteksta.

Nemerle makro sistēma dod iespēju ģenerēt bieži atkārtojāmo kodu bez programmētāja piepūles. Tas statiskā tipizācija ļauj kompilatoram izpildīt statiskas ģenerētā koda pārbaudes kompilācijas laikā. Tas kopumā dod iespēju programmatiski ģenerēt pārbaudāmu un tipu korektu kodu.

Nemerle realizētā makro sistēma ir ļoti līdzīga aprakstītai transformāciju sistēmai, kaut arī tā ir ļoti spēcīga un ļauj izpildīt daļēju novērtēšanu, tā ir ļoti atkarīga no valodas specifikas. Tā kā Nemerle ir statiski tipizēta, tā dod iespēju kompilatoram pārbaudīt kompilēto kodu, nevis ķert tipu kļūdas programmas izpildes laikā. 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 Nemerle mājaslapā [19].

[1] Sakarā ar sintakses īpašībām pēc filmas ”Zvaigžņu kari” iziešanas uz ekrāniem, parādās joks par maģistra Jodas runas stilu - ”The mistery of Yoda’s speech uncovered is: Just an old Forth programmer Yoda was”.

4.5. OpenZz

OpenZz parsētājs ir interpretējams dinamisks parsētājs, kas ļauj ātri izstrādāt parsēšanas risinājumus. OpenZz ļauj modificēt un paplašināt parsētās valodas gramatiku lietojot komandas tajā pašā valodā. 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, jo valodā ir jāievieš speciāli modificēšanas mehānismi. Arī integrācija ar jau eksistējošiem kompilatoriem varētu būt problemātiska.

Diemžēl šīs parsētājs netiek attīstīts kopš 2002. gada un informācija par to ir ļoti ierobežota. Sīkāka informācija par šo rīku ir dabūjama rakstā [5], kas apskata parsētāja koncepciju, un parsētāja mājaslapa [12].

REZULTĀTI

Šajā darbā tiek definēta programmēšanas valodu paplašināšanas sistēmas koncepcija, kas varētu tikt ieviesta jebkādai LL-parsējamai valodai. Sistēmas ideja ir radusies, izstrādājot kompilatoru valodai Eq, un visticamāk tālākos izstrādes posmos sistēma tiks integrēta Eq kompilatorā.

Transformācijas sistēma ir iedvesmota ar divu koda apstrādes principu kombināciju – ar dinamisko parsēšanu un koda priekšprocesēšanu. Dinamiskas parsēšanas pamatprincips ir valodas gramatikas modificēšana koda parsēšanas laikā, tomēr šai pieejai vispārīgā gadījumā piemīt ļoti mazas valodas gramatikas kontroles iespējās un tās darbam ir nepieciešams specifisks parsētāja modelis, kas ļauj darba laikā modificēt savu parsēšanas tabulu. Vispārpieņemtais priekšprocesēšanas princips ir programmas teksta apstrāde bez iedziļināšanās programmēšanas valodas struktūrā, toties dažās situācijās šāda pieeja ir nepietiekama nopietnām izejas koda modifikācijām.

Sistēmas projektēšanā tika piedāvātas pieejas, kas ļaus izvairīties no abu principu trūkumiem. Sistēma tiks veidota ka virsbūve valodas parsētājam, ielasot noteiktas sintakses makro izteiksmes, kas sastāvēs no tipu aprakstiem, regulāro izteiksmju šablona un transformācijas funkcijas. Sistēma strādās paralēli ar parsētāju, pirms vai pēc katras parsētāja produkcijas pārbaudes izpildot nepieciešamas virkņu transformāciju. Programmas teksti tiks apstrādāti daļiņu veidā, nepieciešamības gadījumā vēršoties pie parsētāja pēc papildus informācijas. Tomēr šī sistēma tiek projektēta neatkarīgi no parsētāja, pieprasot minimālas zināšanas par valodas gramatiku, proti, iespējamo valodas daļiņu un pseido-daļiņu tipus. Tipu zināšana ir nepieciešama, lai ieviestu noteiktu transformāciju korektuma kontroli makro ielasīšanas brīdī.

Šī darba ietvaros tika izstrādāts prototips sakrišanu meklēšanas sistēmai. Darba izstrādes gaitā tika izveidota algoritmu kopa, kas ļaus efektīvi meklēt daļiņu virkņu sakrišanas ar šabloniem, un izvēlētie algoritmi tika implementēti prototipā. Lai pēc iespējas samazināt šablonu sakrišanu meklēšanas laiku, tika izvēlēts lietot minimizētus determinētus galīgus automātus katra šablona reprezentēšanai. Vēl lielākai apstrādes laika samazināšanai tika izvēlēts apvienot ielasīto regulāro izteiksmju automātus vienā kopīgā automātā. Uz šīs pieejas bāzes tika izveidoti makro šablonu konfliktu risināšanas un programmas tvērumu maiņu apstrādes principi.

Prototipa izstrādes laikā tika identificētas iespējamās problēmas un izņēmumi, kuri ierobežo šablonu sistēmas darbu. Lietojot izvēlēto pieeju, nebūs iespējams grupēt šablonu daļas ar iekavām un adresēt regulāro izteiksmju apakšgrupas, jo automātu determinēšana un minimizēšana pārkārto grupējumus. Sistēma arī nevarēs patstāvīgi apstrādāt daļiņu klašu mantošanās, tam būs nepieciešama saskarne ar parsētāju. Izstrādāto prototipu reālās sistēmas implementācijas gaitā varēs optimizēt, nemainot darbības principu, bet aizstājot izvēlētos algoritmus ar optimālākiem.

Darbs parāda, ka ir iespējams veidot regulāro izteiksmju šablonus un ar to palīdzību attiecīgi meklēt sakrišanas ieejas daļiņu virkni. Gadījumā, ja valodas parsētājs būs modelēts aprakstītā veidā, būs iespējams veikt sakrišanu meklēšanu. Izstrādātais prototips var kļūt par pamatu tālākai transformācijas sistēmas prototipa izstrādei un implementēšanai reālā kompilatorā. Šablonu apakšsistēmās atrastās virknes kalpos ka ieejas dati transformācijas funkcijai. Tipu sistēma, sa-

vukārt, lietos regulāro izteiksmju minimizētos automātus, lai pārbaudītu tipu sakritību makro izteiksmju ietvaros.

Darba gaitā tika izpētītas eksistējošās izejas koda apstrādāšanas sistēmas, to īpašības un atšķirības no izvēlētās pieejas. Lielākai meta-programmēšanas sistēmu daļai ir ļoti cieša saistība ar konkrētas programmēšanas valodas implementācijas īpašībām (piem. Lisp un Forth). No otras puses, universālie priekšprocesori apstrādā programmu tekstu, neievērojot valodas struktūru. Aprakstītā sistēma, savukārt, ir pielāgojama dažādām valodām un ar pieeju pie noteiktās saskarnes implementē saikni ar valodas semantiku.

Darba gaitā arī tika sagatavots publikācijas melnraksts, kas apraksta sistēmas koncepciju un pielietošanu. Tālākā sistēmas izstrādes gaitā tas tiks pilnveidots un publicēts. Raksta melnraksts ir atrodams 5. pielikumā.

SECINĀJUMI

Šajā darbā tiek skarts jautājums par dinamiskas makro-apstrādes sistēmas izstrādes iespējamības teorētisku pamatojumu un koncepciju. Darbā arī tiek apskatītas eksistējošo sistēmu nepilnības un parādītas pieejas, kas varētu palīdzēt šo nepilnību risināšanā.

Prototipa izveidošana parāda, ka šablonu sakrišanu meklēšanas sistēma ir implementējama, un var kļūt par pamatu tālākai sistēmas funkcionalitātes prototipēšanai, testēšanai un attīstīšanai.

Uz doto brīdi sistēmas izveide ir pamatota uz konkrētas parsētāju arhitektūras, bet atšķirībā no lielākas līdzīgo darbu daļas, tā nav piesaistīta konkrētai programmēšanas valodai. Tālākos izstrādes posmos tās arhitektūra var tikt pielāgota arī citiem parsētāju modeļiem.

Makro sistēmas izstrāde tiks turpināta un aprakstītai sistēmai tiks implementēts reāls lietojams piemērs. Ir saprotams, ka pilnās sistēmas izstrādes gaitā tiks atrastas jaunas problēmas, kuras būs jārisinā, un jaunas idejas, ko varēs pielietot realizācijā. Neskatoties uz to, pirmais izstrādes solis ir izdarīts un ir cerams, ka šādas transformācijas sistēmas projekts būs noderīgs programmēšanas valodu izstrādes, kompilatoru optimizēšanas un makro valodu jomās.

Pielikums 1 PROTOTIPA KODA PIEMĒRS

```
class matcher (object):
    """ The matcher system class. """

    def __init__ (self):
        self.getter = getter ()
        self.parser = regexp_parser (self.getter)
        self.regexp_id = 0
        self.current_level = 0

        self.automata_stack = []
        self.automaton = None

        self.regexp_priority_stack = []
        self.regexp_priority = contexted_id_list ()

    def enter_context (self):
        """ Enters a context, saves the state to be able to return to the
            previous level. """
        self.automata_stack.append (self.automaton)
        self.regexp_priority_stack.append (self.regexp_priority)
        self.current_level += 1
        print "Entered a context, level", self.current_level

    def leave_context (self):
        """ Leaves a context, restores the state. """
        self.automaton = self.automata_stack.pop ()
        self.regexp_priority = self.regexp_priority_stack.pop ()
        self.current_level -= 1
        print "Left a context, level", self.current_level

    def add_match (self, regexp):
        """ Adds a match to the current context. """
        self.getter.set_stream (regexp)
        auto = self.parser.parse (self.regexp_id)
        self.regexp_priority.add (self.current_level, self.regexp_id)
        self.regexp_id += 1
        if self.automaton is None:
            self.automaton = merge (self.current_level, None, auto[0])
        else:
            self.automaton = merge (self.current_level, self.automaton[0],
                                    auto[0])
        print "Added a regexp, level", self.current_level, ", id", \
            self.regexp_id - 1

    def get_accepting (self, new_state, current_accepted):
        """ Finds the accepted match with the highest priority.
```

```
47              Here new_state is the state in which the automata is that might be
48              the new acceptance state. current_accepted is the currently accepted
49              regexp id for the current length of the matched sequence. If
50              current_accepted is None, it means that there are no accepted
51              matches at this point. The new state is checked for the regexp with
52              the highest priority, taking into account the current_accepted. """
53          cur_best = self.regexp_priority.list.index(current_accepted) \
54              if current_accepted is not None else None
55          for state in new_state.state_list:
56              if state.accepting:
57                  ind = self.regexp_priority.list.index(state.regexp_id)
58                  if cur_best is None or ind > cur_best:
59                      cur_best = ind
60          if cur_best is not None:
61              return self.regexp_priority.list[cur_best]
62          return None
63
64      def match_stream(self, stream):
65          """ Tries to match the collected regexps to the beginning of the given
66              stream. Matches the longest sequence. If several regular expressions
67              accept the same sequence, matches the one with the highest priority
68              according to the order of arrival and context. """
69          if self.automaton is None:
70              # if there's nothing to match - we leave
71              return None
72
73          self.getter.set_stream(stream)
74          token = self.getter.get_token()
75          current_state_list = []
76          current_state_list.append((self.automaton[0], []))
77          accepted_regexp = None
78
79          while token is not None:
80              new_state_list = []
81              current_accepted = None
82              for state, processed_token_list in current_state_list:
83                  # there might be several states active at the same time, as
84                  # there might have been several available moves on the
85                  # previous symbol.
86                  move_list = filter(lambda (x, y): x.le(token),
87                                     state.paths.iteritems())
88                  new_token_list = processed_token_list[:]
89                  new_token_list.append(token)
90                  for move in move_list:
91                      # there might be several moves available because we
92                      # allow adding values to the regular expressions, e.g.
93                      # {id} and {id:foo}.
94                      new_state = move[1]
95                      current_accepted = self.get_accepting(new_state,
```

```
96                                                      current_accepted)
97                  new_state_list.append((new_state, new_token_list))
98                  if current_accepted is not None:
99                      # if we found a longer or a priority match, we
100                     # make it the temporary result.
101                     accepted_regexp = (current_accepted,
102                                        new_token_list)
103         current_state_list = new_state_list
104         token = self.getter.get_token()
105     if accepted_regexp is not None:
106         print "\tAccepted regexp", accepted_regexp[0], accepted_regexp[1]
107         return (accepted_regexp[0], len(accepted_regexp[1]))
108     return None
```

Pielikums 2 TESTU KODA PIEMĒRS – TVĒRUMU TESTS

```
def test_case_1 ():
    """ A test for context switching and matching (depth = 1). """
    regexp_1 = [t_m_start(), t_id(), t_lbrace(), t_real(), t_m_asterisk(),
                t_rbrace(), t_m_end()]
    # {match} {id} {(} {real} * {)} {\match}

    list_1 =   [t_id('f'), t_lbrace(), t_real(3.5), t_rbrace()]
    # f(3.5) - Should match #0

    regexp_2 = [t_m_start(), t_id(), t_lbrace(), t_real(), t_rbrace(),
                t_m_end()]
    # {match} {id} {(} {real} {)} {\match}

    list_2 =   [t_id('g'), t_lbrace(), t_real(5.78), t_rbrace()]
    # g(5.78) - Should match #1

    list_3 =   [t_id('e'), t_lbrace(), t_real(4.0), t_real(5.8), t_rbrace()]
    # e(4.0 5.8) - Should match #0

    list_4 =   [t_id('d'), t_lbrace(), t_real(3.4), t_rbrace()]
    # d(3.4) - Should match #0

    transform_system = matcher ()
    transform_system.add_match (regexp_1)
    transform_system.enter_context ()
    assert transform_system.match_stream (list_1) == (0, 4)
    transform_system.add_match (regexp_2)
    assert transform_system.match_stream (list_2) == (1, 4)
    assert transform_system.match_stream (list_3) == (0, 5)
    transform_system.leave_context ()
    assert transform_system.match_stream (list_4) == (0, 4)
```

Pielikums 3 TESTU KODA PIEMĒRS – PRIORITĀŠU UN GARUMU TESTS

```
def test_case_7 ():
    """ A test for checking specific matching (with token values) and the
        longest match. """
    regexp_1 = [t_m_start(), t_id('foo'), t_int(), t_delim_col(), t_m_end()]
    # {match} {id:foo} {int} {;} {\match}

    regexp_2 = [t_m_start(), t_id(), t_int(), t_delim_col(),
                t_int(), t_m_asterisk(), t_m_end()]
    # {match} {id} {int} {;} {int} * {\match}

    list_1 = [t_id('foo'), t_int(4), t_delim_col()]
    # foo 4; - Should match #0

    list_2 = [t_id(), t_int(4), t_delim_col()]
    # {id} 4; - Should match #1

    list_3 = [t_id('foo'), t_int(4), t_delim_col(), t_int(5)]
    # foo 4; 5 - Should match #1

    transform_system = matcher ()
    transform_system.add_match (regexp_1)
    transform_system.add_match (regexp_2)
    assert transform_system.match_stream (list_1) == (0, 3)
    assert transform_system.match_stream (list_2) == (1, 3)
    assert transform_system.match_stream (list_3) == (1, 4)
```

Pielikums 4 TESTU KODA PIEMĒRS – TVĒRUMU IEEJAS UN IZEJAS TESTS

```
def test_case_8 ():
    """ A test for checking contexts (are there any phantoms left after leaving
        a context). """
    regexp_1 = [t_m_start(), t_id(), t_m_end()]
    # {match} {id} {\match}

    regexp_2 = [t_m_start(), t_id(), t_m_asterisk(), t_m_end()]
    # {match} {id} {\match}

    list_1 = [t_id('foo'), t_id()]
    # foo - Should match #0

    list_2 = [t_id('bar'), t_id()]
    # bar - Should match #1

    transform_system = matcher ()
    transform_system.add_match (regexp_1)
    transform_system.enter_context ()
    transform_system.add_match (regexp_2)
    assert transform_system.match_stream (list_1) == (1, 2)
    transform_system.leave_context ()
    assert transform_system.match_stream (list_2) == (0, 1)
```

Pielikums 5 SISTĒMAS APRAKSTA MELNRAKSTS

Šī pielikumā ir atrodams melnraksts rakstam, kas tika izveidots darba izstrādes laika. Raksts pagaidām netika publicēts un atrodas izveidošanas posmā.

On dynamic extensions of context-dependent parsers — Extended Abstract —

Jūlija Pečerska[1], Artjoms Šinkarovs[2], Pavels Zaičenkovs[3]

[1] University of Latvia, Raiņa bulvāris 19, Rīga, Latvija, LV-1586
[2] Heriot-Watt University, Riccarton, Edinburgh, EH14 4AS, United Kingdom
[3] Moscow Institute of Physics and Technology, 141700, 9, Institutskii per., Dolgoprudny, Moscow Region, Russia

Abstract. Modern programming languages define a syntax that cannot be described precisely using context free grammars. This problem is well known, and as a solution there are parser generators like ANTLR, which generates LL(*) parsers rather than LALR/LR. Nevertheless, a lot of real-world parsers are implemented by hand. A great difficulty arises when there appears a need to introduce some changes to the grammar of the language. This implies that the parser has to be rewritten to include new rules and resolve the created ambiguities manually.
Whenever radical changes to the language are required, it is understood that parser rewriting will be necessary. However, when syntax requires some extensions to the convenience of the user, it is natural to allow her to adapt the existing syntax of the selected programming language to her needs. In this paper we present a way of building a system that allows introducing dynamic changes to the grammar and that works on top of any recursive descent parser that meets some minor restrictions.
The execution of the proposed system consists of 2 steps. First is matching a list of tokens against a regular expression built on tokens and encoded production names of the given grammar. And the second is transforming the result using a functional language. We demonstrate that using this approach makes it possible to i) give static guarantees regarding the transformation rules; ii) safely express non-trivial syntactical constructions; and iii) perform a restricted partial evaluation.

1 Introduction

Very often languages like C/C++ introduce new constructs in their syntax. Some of the constructs require serious modifications of the compiler to be supported properly, but some of them may be purely syntactical, like, for example, user-defined literals in C++11 [6]. In this paper we are concerned with a system that allows a class of modifications at the level of the programming language itself, without the necessity to modify the compiler.

The necessity to change the compiler in order to support syntactical changes comes from the fact that most of the programming languages have restricted

capabilities for changing syntax. Some of them do not provide any at all. We believe that the key aspect to the proper self-modifying language is an ability to change a grammar on the fly. If so, one can say that the reasonable approach might be to create a cross compiler which would transform the desired syntax into the syntax recognized by some standard compiler. The problem with this approach is, that most of the base-line languages come with a syntax that is very difficult to parse using an automatic tool. As an example consider the following cases.

1. In C, a user can define an arbitrary type using the `typedef` construct [5], which makes it impossible to disambiguate the expression `(x) + 5`, unless we know whether `x` is a type or not. It could be treated as a type-casting of an application of unary plus to `5` to the type called `x`, or a sum of variable called `x` with `5`.
2. If we were allowed to extend C syntax with an infix binary array concatenation operation denoted `++`, and constant-arrays were allowed to be written as `[1, 2, 3]`, we would immediately run into the problem of disambiguating the following expression: `a ++ [1]`. It could mean an application of postfix `++` indexed by `1` or it could be an array concatenation of `a` and `[1]`.

Sometimes the context may influence not only the parsing decisions but also lexing decisions. Consider the following examples:

1. C++ allows nested templates, which means that one could write an expression `template <typename foo, list <int>>`, assuming that the `>>` closes the two groups. In order to do that, the lexer must be aware of this context, as in a standard context character sequence `>>` means shift right operation.
2. Assuming that a programmer is allowed to define her own operators, the lexer rules must be changed, in case the name of a new operator extends the existing one. For example, assume one defines an operation `+-`. It means that from now on an expression `+-5` should be lexed as (`+-`, `5`), rather than (`+`, `-`, `5`).

In order to resolve the above ambiguities using a grammar-based parser generator, we have to make sure that one can annotate the grammar with correct choices for each shift/reduce or reduce/reduce conflict. This puts a number of requirements on the syntax of a parser generator and on the finite-state machine execution engine. Firstly, one has to introduce contexts without interfering with the above conflict-resolution. Secondly, one has to have an interface to the lexer, in case lexing becomes context-dependent, and all the mechanisms should be aware of error-recovery facilities.

Having said that, we can see that using automatic parser generators could be of the same challenge as writing a parser manually, where all the ambiguities could be carefully resolved according to the language specification. As it turns out, most of the real-world language front-ends use hand-written recursive descent parsers that specially treat cases that cause ambiguity. For example the following languages do: C/C++/ObjectiveC in GNU GCC [2], clang in LLVM [8], JavaScript in Google V8 [3].

That indicates that the first step of cross-compilation would be actually a re-creation of the parser of the base-line language. As it requires a major undertake to create such a grammar, and afterwards one has to update the grammar with the changes made in the original parser, we do not seriously consider this approach. Instead we are going to design a system, which uses an existing parser as a base-line and introduces a set of handles to modify the grammar on the fly. However, arbitrary changes of the grammar may lead to an uncontrolled language evolution which would be hard to verify, so in our approach we allow a restrictive set of changes to the grammar and use a specially designed type-system to control the correctness of transformations.

Very similar tasks, maybe except the correctness verification, are performed by generic macro-processors. Now, is it possible to build our system on one of the existing preprocessing engines, preferably providing some correctness guarantees?

The main goal of any preprocessor is to perform a substitution of one element sequence with another. The unit of the sequence may be different depending on the agreement, however the common case is to say that the unit is a sequence of characters of the same class. The number of classes is normally fixed, however character belonging to the class may be static as in C preprocessor [7], where, for example, notion of space cannot be changed, or dynamic as in TEX, where one could specify that a certain character is a delimiter. Then the substitution itself is a replacement of units in a sequence, which are treated as arguments, with the assigned arguments. The key problem here, as we are concerned in this paper, is a lack of separation between the rewriting itself and the transformation of a token-sequence. Consider an example of a C macro:

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

First of all, it is really hard to say anything about the result of this macro, as `foo (5,6)` expands to `5 6`, but both `foo (,5)` and `foo (5,)` expand to `5`. Secondly, as comma is a part of syntax definition of the macro then one cannot just pass a token sequence `5, 6` as a first argument of `foo`. In order to resolve this one may escape the comma by wrapping it in parentheses and calling `foo ((5,6), 7)` which will expand to `(5, 6) 7`. The only way to flattern the list is to perform an application of another macro. For example:

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

So we have a higher-order macro here, but it would work only if arguments have a right type and the application of `bar ((5,6), x)` would expand to `5, x`, however application of `bar (5, 6)` would not provide an error but would expand to `first 5 6`. And as a last example we can make original macro `foo` return 3 arguments by expanding `foo (5, foo (6, 7))` which will expand to `5 6 7`. We may clearly see that making some static conclusions by checking a system of macros is impossible, as it may all depend on the application; and making any dynamic decisions with respect to the correctness of substitution is also not possible, as there is no way to declare the criteria of correctness.

Despite all the correctness complications macro systems are not powerful enough to introduce new language constructs. For example, it would be natural to represent a number's absolute value as `|a|`, or to allow a number of user-defined literals to introduce units in a programming language like `5kg` or `8 mm`. Even if a macro-system can do it, resolving nested expressions treating one and the same symbol differently still might be confusing. For example, would it be possible for some macro system to transform an expression `| a|b |` into `abs (a|b)`?

So we just saw that using a preprocessor as it is for the desired task does not exactly meet our intentions. As a solution to the given problem we introduce a transformation system which works on the sequence of pseudo-tokens, which is a combination of tokens recognized by parser and productions of the grammar used by parser. To make it even more powerful we allow a regular expression on pseudo-tokens, still being able to guarantee the correctness of the transformation.

The rest of the paper is organized as follows: we are going to introduce a basic model of the parser which we use to build a transformation system on in section 2. Then we describe the syntax of the transformation rules and the way we intend to prove correctness in section 3. Section 4 describes the way we deal with multiple transformation rules and finally we evaluate and conclude the work in sections 5 and 6.

2 Parser model

The parser which serves as a basis for building a transformation system is based on recursive descent LL(k) or LL(*) algorithm. Recursive descent is a natural human approach to writing parsers, and in case if k is small, the efficiency of parsing is linear with respect to the number of tokens on the input stream.

As a running example in this paper we are going to use a grammar of a simple language with C-like syntax described in Fig. 1.

```
program       ::=  ( function ) *
function      ::=  type_id  '(' arg_list ')' stmt_block
arg_list      ::=  ( type_id  id ) *
stmt_block    ::=  '{' ( expr | return ';' ) * '}'
expr          ::=  fun_call | assign | cond_expr
fun_call      ::=  id '('expr (',' expr ) * ')'
assign        ::=  id '=' expr
cond_expr     ::=  bin_expr ( '?' cond_expr ':' expr ) *
bin_expr      ::=  primary_expr ( binop primary_expr ) *
primary_expr  ::=  number | prefix_op expr | '(' expr ')'
binop         ::=  '&&' | '||' | '==' | '!=' ...
prefix_op     ::=  '-' | '+' | '!' | '~'
```

Fig. 1. A grammar of a C-like language.

As the transformation system is built as an extension to the parser, it expects a certain behaviour of the parser. Further down we list a set of properties we require to be present in the implementation of the parser.

Token stream The parser should conceptually represent a stream of tokens as a doubly linked list, which allows traversing in either direction, performing a substitution of a token group with another token group, and restarting a stream from an arbitrary position. The implementation details are left to the creators of the parser.

Pseudo-tokens The parser normally reduces the grammar rule by reading tokens from the input stream. We introduce a notion of pseudo-token, which conceptually is an atomic element of the input stream, but that represents a reduced grammar rule. The implementation details are left to the parser creator. The most straight forward and inefficient way would be to convert the pseudo-token back into the token stream and parse again.

Handle-functions First of all, we ask that every production is represented as a function[4] with a signature `Parser` $\rightarrow$ `(AST|Error)`, i.e. function gets a parser-object as an input and returns either an Abstract Syntax Tree node or an error. We call those functions handle-functions. We require that handle-function structure mimics the formulation of the grammar, i.e. if a production A depends on a production B, we require function handle-A to call function handle-B.

Each handle-function implements error recovery (if needed) and takes care of disambiguating productions according to the language specification, resolving operation priorities, syntax ambiguities and so on. Each handle function has access to the parser, which keeps record of an internal state, which changes when a handle-function is applied.

Is-functions Each handle-function is paired with a predicate function which checks whether a sequence of tokens pointed at by a parser state matches a given rule. We will call this type of functions is-functions. Application of an is-function does not modify the state of the parser. Is-functions may require unbounded look-ahead in general case, however we leave the implementation decision to the parser creator. One can always reuse matched AST nodes to perform subsequent matches.

Match-function In the beginning of each handle-function, each production calls a function called `match` with a signature `(Parser, Production)` $\rightarrow$ `Parser`. A match-function is an interface to the transformation system that checks if a stream of tokens pointed at by the parser has a valid substitution in the given production; if it does, it performs the substitution and makes sure that the parser points to the beginning of the substitution in the token stream. In case if no matches were found, the transformation system does not perform any substitutions and returns the parser in its original state.

Assuming that all the described requirements are met, the grammar $G = (N, T, P, S)$ provides complete information required to create support for user-defined matches.

[4] Note that these functions have side-effects, so the order of calling is important.

3 Transformation system

In this section we are going to describe a syntax of the rules of the transformation system and demonstrate a way to prove the correctness of the transformation.

3.1 Match Syntax

The transformation system consists of the *match* rules and token-transformation functions. First of all we would consider the match rule, which modifies a behaviour of a given production. The syntax of the rule can be learned from the following example.

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

This reads as follows: if at the beginning of production `prod1` a stream of pseudo-tokens pointed at by the parser matches a regular expression `regexp`, which can be aliased with variable named `v` in the right hand side of the match, then the matched tokens will be replaced with a reduction of `prod2` production applied on a list of pseudo-tokens that is being returned by `f (v)`. Function `f` is a function which is defined in functional language T and which is used to perform a transformation on the list of tokens matched by the left hand side of the match.

The `regexp` regular expression is a box standard regular expression [9] which is defined by the grammar at Fig. 2. In this paper we are using a minimalistic

```
regexp          ::= concat-regexp '|' regexp
concat-regexp   ::= asterisk-regexp  concat-regexp
asterisk-regexp ::= unary-regexp '*' | unary-regexp
unary-regexp    ::= pseudo-token | '(' regexp ')'
```

Fig. 2. Grammar of the regular expressions on pseudo-tokens

syntax for regular expression to demonstrate some basic properties. Later on, this syntax may be easily extended.

Further down in this paper we are going to use an escaped syntax for pseudo-tokens which represent grammar production names, like `\expr`, `\function`, etc. The operators of regular expression, namely (`|`, `*`, `(`, `)`) will be escaped as well like: (`\|`, `\*`, `\(`, `\)`). For simplicity reasons we assume that we do not introduce any conflicts by defining escape symbols. In case a conflict appears, we may change the escape symbol, or even the whole escaping mechanism, but it does not influence the matter.

Now we can demonstrate a simple substitution example on the language defined in Fig. 1. Assume that function `replace` is defined in T with three arguments and in any list of pseudo-tokens it replaces each ocurrence of the second argument with the third, and what we need is to call a function called `bar` with an argument being the summed-up arguments of function called `foo`. In that case the following match would perform such a substitution.

```
match [\fun_call] v = foo ( \expr \( , \expr \) \* )
   -> [\fun_call] cons bar (replace (tail v) \, \+)
```

3.2 Definition of matches

Match rules can be defined in arbitrary places of a program and the rule activates immediately after the definition was parsed. We do however differentiate between the global matches and context matches. In our case the context is created by a `stmt_block` production. In that case, all the matches declared within the `stmt_block` production are valid only within this particular production. When the production is finished, the matches declared within the production would be removed. Declaration of the context is up to the parser, it may define it in any way by calling two interface functions. The context definition can be omitted in which case all the matches would be global.

Both global and context matches depend on the order of definition and in both cases the first match has a stronger priority than the subsequent in case the token stream can be matched with more than one regular expression. The priority helps to cover the following case:

```
match [\expr] v = f ( \num )            ...
mathc [\expr] v = f ( \primary_expr )   ...
match [\expr] v = f ( \expr )           ...
```

Here we can see that the last regular expression includes the previous as `\primary_expr` is also an `\expr` and so on. But by introducing priorities one may still have a different behaviour in each case.

Nested context matches overload the outer matches, in a same way as local variables have a first match, in case a variable of the same name exists in the outer context.

3.3 *T* language and correctness

In order to perform a transformation of the matched list we define a minimalistic functional language called T to demonstrate the approach. The core definition of T is given by Fig. 3.

The main use-case of the language is to traverse over the matched list of pseudo-tokens applying recursion, head, tail and cons constructs. In order to stop the recursion we also introduce arithmetic operations on integers. In order to perform partial evaluation, we need to have an interface to get the value of a pseudo-token. For that reason we introduce function `value` which is applicable to the pseudo-tokens which have a constant integer value (in our example it is a `\number` pseudo-token). In order to construct an object from integer, we are using `\number[42]` syntax. The `value` function operates on integers only for the simplicity of the model, but the basic types can be extended in future.

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

Fig. 3. Grammar to define language T

Type system In order to prove the correctness of the match, we are going to use a specially designed type-system which is based on the regular expressions being treated as types. We are going to use a type inference to check if the function application in the right hand side of the match is allowed within a given production. In the current paper we are not going to give a definition of all the type deduction rules, however, we are going to do that in the full paper. Further in this section we are going to share the basic ideas and principles.

The main idea of the type system for T is to treat a regular expression as a type. We start with a fact that if the right-hand side of the system was called and the match succeeded then the result of the match is a flat list of pseudo-tokens. However from the regular expression we have additional information about the structure of this list. In order to perform a type inference, we observe that regular languages, hence regular expressions, bring a number of set operations, which is the key driving force of the inference. First of all, it is easy to define a subset relationship on two regular expressions $r_1 \sqsubseteq r_2$. As we know, we can always build a DFA for a regular expression and minimize it [1], which gives us a minimal possible automaton for the language recognized by a given regular expression. It means that $r_1 \sqsubseteq r_2 \Rightarrow min(det(r_1)) \sqsubseteq min(det(r_2))$. For two minimized automata A_1 and A_2, $A_1 \sqsubseteq A_2$ means that there is a mapping Ψ of A_1 states to A_2 states such that:

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

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

$States(x)$ denotes a set of all the states of automaton x, $Edges(s)$ is a set of pseudo-tokens which mark the outcoming edges of state s. Finally, $Transition(s, t)$ denotes a state which is reachable from s, using edge marked with t.

The subset relationship is going to be used to create a sub-typing hierarchy, and we also have a notion of a super-type of the hierarchy, which is represented by a regular expression `.*`, we are going to denote this type $\top$. It is obvious to see that $\forall t_i \in R, t_i \sqsubseteq \top$.

It is possible to construct a type for head and tail application by following the edges from the starting state of DFA in case of head, and creating a set of sub-automata in case of tail. Furthermore we can infer a type for recursive head/tail traversal over the list in either direction. In order to do that, one has to construct a set of all the sub-automata of a given one in case of forward traversal and the set of all sub-automata of the reversed automaton in case of backward.

In order to propagate type information in the branches obtained from the application of `type` construct, we have to know how to subtract two regular expressions [10], which is, again, simple. If T and S are respective DFA for subtracted types, all we have to do is to construct $C = T \times S$ and make the final states of C be the pairs where T states are final and S states are not.

The type inference procedure itself borrows the idea from inferring the shape of the array in SaC type system [4] i.e. we will start with an abstract type `regexp_t` which is $\top$ and recursively precise the type until we get to a fixed point.

4 Regular expressions

We have devised a method of implementing match syntax that will be time effective during the transformation. We have also created a prototype that ilustrates some of the features described in this article. This section will, in short, describe the approach we selected, for the full description, however, turn to the full paper.

As it was said, match syntax is a simple regular expression. This expression is parsed and transformed into a non-deterministic finite automaton (NFA). Next step is to create a deterministic finite automaton (DFA) out of the created NFA to minimize the effort needed to execute the match. This task is performed using the subset construction algorithm.

In order to minimise execution time even further, we minimise the created DFA. The minimisation algorithm is described in detail in the "Dragon Book" [1]. The algorithm creates sets of states that cannot be distinguished by any input token sequence. Once the algorithm fails to break the sets into smaller ones it stops. These sets of states then become new states of the minimal automaton.

It is essential to note that the existence of such minimal automaton is provable, despite the complexity of the regular expression it describes. This statement implies that we have a possibility to prove equality of two automata. As the naming of the states is unimportant, we will say that two automata are the same up to state names if one can be transformed to another be simply renaming the states. Therefore two regular expressions match the same input if and only if their automata are the same up to state names.

The construction of the DFA does not support back-referencing, so the bracket groups are intended only to change the priority of the operations in the regular expressions. We abandon the back-referencing feature in favor of maintaining a fully determinate automaton for each regular expression.

4.1 DFA grouping

As we operate with source code, we want to check all of the matches that we have in one pass through the code. We consider two possibilities to merge the created automata into a match system in order to improve effieciency. For both of them we evaluate DFA adding, matching and context inheritance algorithmic difficulty in the main article.

Context inheritance difficulty is important, as we can have several included contexts, where the matches introduced inside the included one have a higher priority. Imagining that we have a system of `n` automata we have two options of doing so. The first one is that on entering a context we add the `m` matches we come across to the main match set and remove the `m` matches upon exiting the context. The second one is that on entering a context we create a copy of the parent context, to which we add the `m` newly found matches and upon exiting the context throw out the entire new system of matches.

First of the options we consider is joining the automata in a list. Let us assume that there are `n` matches to be checked. Then the automata list represents an NFA with `n` epsilon branches from the start state, each of which leads to one of the DFA we already created. This case is fairly simple and will not be described here extensively.

The second option is more complex. In this case we combine the automata created for each of the matches together into a single DFA. This is done in order to reduce matching time, the automata merging algorithm is described in the full article. The second option is more interesting, as it allows matching of `n` independent patterns in $O(l)$ steps, where `l` is the amount of tokens to be matched. It is important to note as well that our method of joining the automata preserves the priority hierarchy of the matches.

We do not explicitly select a method of operation with joining DFA in the article, we only give the complexities of the proposed variants. This is because the optimal solution selection should be based on the practical uses of the system. Even though the second option is time-consuming when adding and removing matches, the dramatic improvement in execution time might come from the fact that match count is relatively small, but the automaton execution will, at worst cases, be performed for every token in the source text. The actual selection remains to be made by the readers.

5 Evaluation

In this section we are going to consider the common cases for application of the designed approach and evaluate the results.

5.1 Syntax extension

One of the typical examples of using the transformation system is to extend the syntax of the language. Assume that we would like to have a mathematical

notation for absolute value, such as $|5| \equiv abs(5)$. It can be achieved using the following match:

```
match [\expr] v = | \expr |
   -> [\expr] cons (\id[abs]) cons \( cons (head (head v)) cons \) cons nil
```

Keep in mind that we could also define the same match on numbers and in that case we may use the partial evaluation capabilities of the system. Consider the following example:

```
match [\expr] v = | \num |  -> [\num] mabs (head (head v))
mabs :: \num -> \num
mabs x =  if value (x) > 0 then
             \num[(value x)]
          else
             \num[0 - (value x)]
```

The one thing that is left outside the scope of this paper and that is directly connected with this example – do we allow expressions such as `| a|b |`. Currently it is going to be considered as an error, as there is no way to express the fact that one should use the original pseudo-token for the expression inside the absolute value, without applying matches. It depends on the agreement, of course, but currently one can overload a special case for `| \expr \| \expr \* |` to allow the latter. However, in the current model the following would work just fine: `||A| + |B||`, as well as this: `|(a|b)|`.

5.2 C++ Templates

The described transformation system may be used in a similar way as template meta-programming technique. In the case of templates, instantiation process may be delayed until some compiler optimisation finds out that certain template parameter is a constant. In our setting the instantiation happens during the parsing phase and if the parser is unable to infer the value for a match parameter the instantiation will simply not happen. We are going to consider a way to express the factorial function in match syntax that will be evaluated in compile time.

```
match [\number] v = \number ! -> [\number] fact-match (v)
match   [\expr] v = \expr   ! ->   [\expr] fact-match-expr (v)

fact-match :: \number -> \number
fact-match x = if (value x) == 0 then
                    \number[1]
               else
                   \number[(value x)
                            * (value (fact-match
                                      \number[(value x) - 1]))]
```

```
fact-match-expr :: \expr -> regexp_t
fact-match-expr x = cons \id[factorial] cons \( cons x cons \) nil
```

This example demonstrates an advantage over the C++ templates, which is the shared syntactical form of compile-time and run-time match instances. It means that if in C++ the factorial function is defined as a template:

```
template <int n> struct fact
{
  static const int value = n * fact<n - 1>::value;
};

template <>
struct fact<0>
{
  static const int value = 1;
};
```

, there is no way to call it with a non-static variable. The only way would be to create a wrapper function/macro, but in order to find out if something is a constant at compile time one has to have mechanisms similar to `__builtin_constant_p` introduced in GCC v4.7.

5.3 Preprocessing

We describe a method of building a transformation system, which operates in terms of a single production only. C preprocessor, on the other hand, works out of scope. For example, we can use `#if ... #endif` directives wherever in the source code, for example, starting in the middle of one production and finishing in the middle of another, which makes static verification difficult. Using the transformation system defined in this paper it is possible to create a match that would emulate C preprocessor macro-if. It might look as follows:

```
match [\p1] v = # if expr  \. \* # endif  -> [\p2] ...
```

The dot syntax is currently not allowed, but it is possible to emulate it using a disjunction over all of the pseudo-tokens. Now, depending on what is written in the right hand side of the match, it may be accepted by the type system or not. In case the type-system can prove that the result of the right-hand side expression could be recognized by `\p2`, it is allowed. However, it is easy to understand that there is always a way to make the type-system happy, i.e. by generating a constant expression. It means that the macro-if can be emulated. We will leave the discussion open on the question whether it is good or bad. In order to avoid this situation one may declare stricter type-checking rules.

6 Conclusion

This paper describes a theoretical background for constructing an extensible dynamic macroprocessor and provides a view on solving shortcomings that existing preprocessors have. We present to the judjment of the reader a parser model that includes facilities necessary for the dynamic extension constructing and the syntax we propose for describing macros using regular expression elements. We also give several examples to demonstrate the described extension's capabilities, which can be applied to perform metaprogramming techniques as well as language syntax extension.

We have prototyped the described system and plan to develop it further on to create an actual usable example of the described approach. There are, for sure, still plenty of issues to tackle and resolve, however, we expect the project to be helpful in wide areas of language design, compiler optimisations and macro languages.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools, Second Edition.* Addison-Wesley, 2007.
2. Free Software Foundation. GCC. http://gcc.gnu.org.
3. Google. V8 JavaScript Engine. http://code.google.com/p/v8.
4. Clemens Grelck and Sven-Bodo Scholz. Sac: a functional array language for efficient multi-threaded execution. *Int. J. Parallel Program.*, 34(4):383–427, August 2006.
5. ISO/IEC. ISO/IEC 9899:1999: Programming languages – C. Technical report, International Organization for Standardization, Geneva, Switzerland., 1999.
6. ISO/IEC. Iso/iec 14882:2011, information technology – programming languages – C++. Technical report, International Organization for Standardization, Geneva, Switzerland., 2011.
7. Richard M. Stallman and Zachary Weinberg. *The C Preprocessor*, 2005.
8. LLVM Team. CLANG. http://clang.llvm.org.
9. Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, June 1968.
10. Sheng Yu. Handbook of formal languages, vol. 1. chapter Regular languages, pages 41–110. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

Literatūras saraksts

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[2] F. Bassino, J. David, and C. Nicaud. On the average complexity of moore's state minimization algorithm. *CoRR*, abs/0902.1048, 2009.

[3] J. Berstel, L. Boasson, O. Carton, and I. Fagnot. Minimization of automata. *CoRR*, abs/1010.5318, 2010.

[4] P. Boullier. Dynamic grammars and semantic analysis. Rapport de recherche RR-2322, INRIA, 1994. Projet CHLOE.

[5] S. Cabasino, P. S. Paolucci, and G. M. Todesco. Dynamic parsers and evolving grammars. *SIGPLAN Not.*, 27(11):39–48, Nov. 1992.

[6] H. Christiansen. A survey of adaptable grammars. *SIGPLAN Not.*, 25(11):35–44, Nov. 1990.

[7] R. Cox. Regular expression matching can be simple and fast. 1 2007.

[8] F. S. Foundation. Gcc. http://gcc.gnu.org.

[9] Google. V8 javascript engine. http://code.google.com/p/v8.

[10] J. Hodgson. Prolog valodas standarts, operatoru definēšana. http://pauillac.inria.fr/ deransar/prolog/bips.html#operators.

[11] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[12] INFN. Openzz mājaslapa. http://openzz.sourceforge.net/.

[13] B. Khoussainov. Regular expressions and finite automata. http://www.cs.auckland.ac.nz/ bmk/se211lectures/lecture14.pdf.

[14] C. H. Moore. Forth mājaslapa. http://www.forth.org/.

[15] S.-B. Scholz. A type system for inferring array shapes in high-level array programs.

[16] P. Seibel. *Practical Common Lisp*. Apress, Sept. 2004.

[17] J. N. Shutt. Recursive adaptive grammars. Master's thesis, Worcester Polytechnic Institute, Aug. 1993.

[18] L. Team. Clang. http://clang.llvm.org.

[19] T. N. Team. Nemerle wiki lapa. http://nemerle.org/wiki/index.php?title=Main_Page.

[20] F. Yu, Z. Chen, Y. Diao, T. V. Lakshman, and R. 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.

Bakalaura darbs ”Ar regulārām izteiksmēm paplašinātu gramatiku dinamiska parsēšana” izstrādāts LU Datorikas fakultātē.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: …… Jūlija Pečerska

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: profesors Dr. dat. Guntis Arnicāns …… 28.05.2012.

Recenzents: profesors Dr. dat. Kārlis Čerāns

Darbs iesniegts Datorikas fakultātē 28.05.2012.

Dekāna pilnvarotā persona: vecākā metodiķe Ārija Sproģe ……

Darbs aizstāvēts bakalaura gala pārbaudījumu komisijas sēdē

07.06.2012. prot. Nr. …

Komisijas sekretāre: ……