Електротехнички факултет Универзитета у Београду
Катедра за Рачунарску технику и информатику

дипломски рад

Унапређење генератора парсера CUP за рад са апстрактним синтаксним стаблима

ментор
др Драган Бојић

студент
Душан Станковић

Београд, октобар 2009.

Захваљујем се свом ментору др Драгану Бојићу и Милошу Глигорићу на помоћи и сугестијама при изради овог рада.

октобар 2009.
Душан Станковић

Садржај

1. Увод 5
2. Преглед коришћених технологија 6
   2.1 Генератор парсера CUP 6
   2.2 Остали алати 7
3. Опис решења 8
   3.1 Опис имплементације CUP генератора парсера 8
   3.2 Опис имплементације додатка ast extension 9
4. Преглед имплементације и коришћени пројектни узорци 12
5. Упутство за употребу алата 14
   5.1 Аутоматско генерисање класа 14
   5.2 Додавање акција за градњу стабла 15
6. Поређење са постојећим решењима 17
7. Закључак 18
8. Литература 19
Додатак А – Пример 20
   А.1 Граматика за једноставне аритметичке изразе 20
   А.2 Граматика са додатим акцијама за грађење стабла 20
   А.3 Пример једне генерисане класе 21
   А.4 Пример посетиоца за рачунање вредности израза 23

Апстракт

Тема овог дипломског рада је унапређење генератора парсера CUP (Constructor of Useful Parsers) [1], додавањем функционалности за рад са апстрактним синтаксним стаблима. Проширење омогућава аутоматско генерисање класа неопходних за градњу апстрактног синтаксног стабла и аутоматско додавање акција за грађење стабла у спецификацију граматике – названог *ast extension*.

Идеја са којом се кренуло у имплементацију овог проширења јесте та да се текст парсера максимално ослободи редундантних информација, тако да корисник генератора парсера сада треба да пише само оно што је неопходно и што дефинише граматику језика за који се жели направити преводилац, а алат сам препознаје и користи све информације о нетерминалима и продукцијама на основу дате дефиниције граматике неког језика.

1. Увод

Апстрактно синтаксно стабло је један од облика представљања програмског кода у облику међукода. Креира га парсер и прослеђује другом делу компајлера за даљу обраду. Представљање кода у неком од међуоблика (*intermediate code*) је погодно јер се тако могу раздвојити предњи и задњи део компајлера и независно имплементирати – нпр. за исти парсер се могу направити независни генератори кода за различите платформе [2]. Такође, ако се користи метод реализације преводилаца уз коришћење апстрактног синтаксног стабла, спецификација парсера коју алат прихвата ће бити много прегледнија. Још једна битна особина апстрактног синтаксног стабла је да оно представља апстрактну синтаксу, односно само оне елементе кода који су битни за очување његове синтаксне и семантичке исправности и информација који он носи (на пример, заграде и специјални карактери, ако нису део неког стринга, су непотребни у стаблу).

За конструкцију апстракног синтаксног стабла потребно је користити класе које представљају нетерминале и продукције, па овај додатак аутоматски генерише све неопходне класе на основу задате граматике. Нетерминали се пресликавају у апстрактне базне класе, а смене у изведене класе из класе терминала који је са леве стране те продукције, тако да се користи особина полиморфизма при обиласку стабла и жељеној обради над њим.

Осим статичког генерисања свих потребних класа, алат има могућност и преуређивања основног текста парсера који садржи граматику језика и аутоматског додавања акција за грађење синтаксног стабла за време рада парсера. Ово је такође значајно за кориснике јер је тај акциони код шаблонски и састоји се само од креирања новог чвора одговарајућег типа на крају препознавања одређене смене, па је стога и најпогоднији за аутоматизовано генерисање, а може да уштеди доста времена кориснику конструктора парсера.

2. Преглед коришћених технологија

Оригиналан Јава програмски код алата CUP, верзија 10k, је послужио као основа за креирање пројекта у развојном окружењу NetBeans, верзија 6.7. Алат CUP је намењен креирању LALR [3] парсера од задатих спецификација уз могућност коришћења уметнутог Јава кода који се у неизмењеном облику пресликава у генерисани парсер, у посебну класу за акције.

2.1 Генератор парсера CUP

Граматика коју прихвата CUP је једноставна и састоји се од смена којима су одређени нетерминални симболи. У сменама могу учествовати нетерминални и терминални симболи који могу имати и тип који је ограничен системом типова програмског језика Јава. Управо ту и јесте могућност за креирање сложених структура као што је апстрактно синтаксно стабло над кодом неког програмског језика, јер нетерминали могу бити типа класа које представљају чворове стабла, а циљ рада овог алата је управо да те класе генерише.

За динамичко креирање апстрактног синтаксног стабла у току рада парсера користи се постојећи систем додавања акционог кода. Алат CUP омогућава пренос вредности кроз смене увођењем променљиве назване RESULT која представља нетерминал на левој страни смене, а унутар једне смене преко лабела које могу бити додељене терминалним и нетерминалним симболима са десне стране неке смене. Пример за креирање новог чвора у неком стаблу је следећи:

```
A ::= B:b C:c {: RESULT = new typeA(b, c) :}
```

где је нетерминал A типа класе *typeA* а симболи B и C могу бити класних или простих типова, то у овом примеру није битно. Статички генерисане класе су такве структуре да у овом примеру класа *typeA* садржи поља која су типа симбола B и C, а сва додела вредности се обавља у конструктору класе *typeA*. Само убацивањем оваквих смена на крај сваке продукције могуће је као резултат рада парсера добити креирано апстрактно синтаксно стабло.

Стабло које парсер креира приликом препознавања улазног текста се гради од листова ка корену стабла, јер је алгоритам рада парсера којим се генерише ово стабло тзв. *bottom-up* парсирање, односно препознавање смена тек када се на улазу појаве сви симболи који дефинишу неку смену. Вредност неког нетерминала, односно чвор којим је он представљен у апстрактном синтаксном стаблу ће касније коришћењем лабела бити прослеђена његовом родитељу у стаблу, и тако све до кореног чвора, односно оног који представља стартни нетерминал. Додавањем акционог кода за генерисање

стабла није могуће створити конфликте у постојећој граматици, јер се они увек додају на крају неке од смена.

2.2 Остали алати

Алат који није директно коришћен при изради овог додатка, али који је послужио као модел чија је функционалност потпуно пренета у имплементацију додатка *ast extension* је *classgen*, верзија 1.4. Он се може користити за генерисање класа било какве структуре и хијерархије, мада је првенствено замишљен као подршка градњи апстрактних синтаксних стабала. Алат *classgen* прихвата спецификацију која врло наликује спецификацији граматике која се користи код CUP генератора парсера, али није идентична. Баш ова чињеница је послужила као идеја за израду дипломског рада и искоришћавање већ написане граматике за добијање свих неопходних информација и генерисање класа. Класе које се генеришу овим додатком у потпуности одговарају онима које генерише *classgen* [4].

Креирање пројекта у развојном окружењу NetBeans уместо програмирања у неком едитору текста и компајлирања из командне линије је одабрано због доста лакшег сналажења са пројектом и напредних могућности које ово окружење нуди – неки од њих су и компајлирање кода у реалном времену уз графички приказ грешака и предлога за њихово исправљање, могућност преименовања променљиве или класе тако да се то одрази на сва појављивања исте у пројекту итд. Структура програмског кода оригиналног алата CUP и распоред класа у пакете су потпуно очувани и приликом креирања пројекта. Развојно окружење NetBeans је слободно доступно и може се наћи на интернету.

3. Опис решења

Овде ће најпре бити приказано како ради постојећа имплементација генератора парсера CUP, које структуре података креира и користи при генерисању парсера, и биће у основном цртама изложен алгоритам рада генератора парсера. Затим ће бити представљено како су неки од тих података искоришћени при конструкцији додатка *ast extension*, а такође ће бити наведене и неке евентуалне модификације и проширења оригиналног кода које су биле неопходне за реализацију овог проширења.

3.1 Опис имплементације CUP генератора парсера

Генератор парсера прво чита улазни фајл који представља спецификацију неке граматике, парсира га и у току парсирања препознаје и памти све симболе и њихове типове, затим све продукције и њихове елементе, а акциони код пресликава у приватну класу за акције где се смешта сав кориснички код. Симболи су представљени класом *symbol*, а из ње су изведене класе *terminal* и *non_terminal*, које представљају терминалне и нетерминалне симболе, респективно. Најбитнији атрибути ових класа су име и тип симбола. Тип који се аутоматски поставља за сваки симбол за који исти није наведен у спецификацији је тип Object. Продукције су представљене класом *production* из које се могу дохватити референце на нетерминал са леве стране продукције, као и на сваки у низу објеката типа *rhs* које представљају симболе са десне стране продукције.

По завршетку парсирања улазне спецификације, могуће је дохватити све препознате терминалне и нетерминалне симболе, као и све продукције. Симболи и продукције се чувају у хеш табелама које се попуњавају приликом препознавања текста спецификације. На основу препознатих симбола и продукција, алат креира интерне табеле и то табелу продукција, табелу редукција и табелу акција. Ове табеле се касније користе при генерисању кода парсера за задату спецификацију. Све табеле су дате у форми стрингова представљених нумеричким литералима, па у том облику нису погодне за евентуалну употребу у циљу коришћења неких информација, као што је то случај са симболима и продукцијама.

Последња етапа у раду генератора парсера је исписивање текста парсера у излазни фајл, при чему се користе све до тада прикупљене информације и генерише класа која представља парсер. У класу парсера се уписују табеле продукција акција и редукција у формату наведеном раније у тексту, а генерише се и приватна класа где се смешта сав кориснички код придружен продукцијама. Тај код се смешта у гране *case* структуре којих има онолико колико има продукција, па је стога прегледан и разумљив. Сам алгоритам парсирања је исти за све парсере генерисане овим алатом и енкапсулиран је класом *lr_parser* коју наслеђују генерисани парсери.

3.2 Опис имплементације додатка ast extension

Користе се структуре података креиране при парсирању спецификације граматике, а које обезбеђује CUP. На основу спецификације сваком симболу се додељује име, тип и лабела. Име је обавезно, тип и лабела нису, и то служи алату да препозна шта је од симбола битно и треба да се нађе у стаблу, а шта не. Пре коришћења тих података пролази се кроз све нетерминале и додељује тип онима за који исти није декларисан. Затим се пролази кроз све продукције и нетерминале и генеришу потребне класе.

На основу података добијених из спецификације граматике статички се генеришу све неопходне класе за грађење апстрактног синтаксног стабла. Постоје три типа класа које граде хијерархију, а заједничко им је да све оне директно или посредно наслеђују заједнички интерфејс *SyntaxNode*:

- Базне класе које представљају нетерминалне симболе. Оне садрже апстрактне методе и никада се директно не појављују у стаблу. Овај тип класе се генерише за сваки препознат нетерминал у граматици који је декларисан типом података који није прост, и за сваки нетерминал за који тип уопште није декларисан (имплицитно Object). Алат претпоставља да је намерно изостављена декларација типа нетерминала, додељује му тип и генерише базну класу
- Изведене класе које представљају појединачне смене. Оне имплементирају све апстрактне методе својих надкласа и основни су елемент при грађењу стабла. Изведена класа се генерише за све продукције неког нетерминала за који је генерисана базна класа, према правилима описаним у претходној тачки. Класа има онолико поља колико продукција има елемената који су битни за стабло. Одбацују се они терминални симболи који нису неопходни за очување семантике кода, која је сада одређена топологијом стабла
- Record класе које су специјални случај изведених класа, али које директно наслеђују интерфејс *SyntaxNode*. Оне се генеришу када неки од нетерминала има само једну продукцију која је названа именом нетерминала. Имају сва поља као и изведене класе, исту функцију, а служе да се смањи укупан број генерисаних класа када извођење из надкласе не доноси никакву корист (полиморфизам нема смисла ако се зна да постоји само једна изведена класа)

Генерисане класе имају следеће методе:

- конструктор који као аргументе прима онолико аргумената колико има поља класе (за базне и Record класе увек постоји и додатно поље које показује на родитеља у стаблу, оно се не поставља у конструктору)
- get и set методе за сва поља класе која су добијена из продукције
- методе за постављање и дохватање родитеља, за оне класе које имају референцу на родитеља

- метода за прихватање посете, и метод којим се деци у стаблу прослеђује посета
- методе за рекурзивни обилазак стабла од врха ка дну, тако што се прво прихвати посета а затим проследи деци, и од дна ка врху тако што се прво проследи деци посета а затим и прихвати
- метода за испис у текстуалном облику (toString)

За модификацију оригиналних фајлова са спецификацијом граматике и додавање акција за грађење стабла коришћен је парсер креиран коришћењем генератора лексичких скенера JFlex и генератора парсера CUP, а све операције над текстом оригиналног фајла врше се акцијама скенера и парсера.

Креира се објекат класе StringBuffer и прослеђује и скенеру и парсеру јер је неопходно да оба уписују у излазни бафер (касније ће то бити излазни фајл). Скенер преписује све терминалне симболе и беле знакове, а парсер додаје текст тамо где је то неопходно. Потребно је да скенер касни са уписом текста у бафер док се не препозна следећи терминални симбол, јер парсер ради тако што гледа један симбол унапред на улазу (lookahead, користи се LALR(1) парсер), па је неопходно обезбедити њихову међусобну синхронизацију.

Први део спецификације који алат модификује је део са декларацијама нетерминалних симбола. Ако за неки нетеркминални симбол није дефинисан тип, препознаје се да је то симбол који треба да учествује у грађењу стабла и додељује му се тип. Ово је неопходно да би се исправно превео код генерисаног парсера јер је при додели вредности променљивој RESULT потребно познавати тип те променљиве, односно нетерминала са леве стране продукције. Назив типа одговара називу нетерминала са великим почетним словом. Такође, неопходно је сада сваку декларацију нетерминала исписати у посебном реду, јер CUP сматра да су сви симболи декларисани један за другим истог типа, што је у реду када тип није дефинисан, али по додели типа више није исправно стање. Пример за ово је следећи:

оригинални текст

```
nonterminal nta, ntb, ntc;
```

модификовани текст

```
nonterminal Nta nta;
nonterminal Ntb ntb;
nonterminal Ntc ntc;
```

Следеће што је потребно додати у текст спецификације јесу лабеле за све терминалне и нетерминалне симболе који се користе у телу неке од смена, да би парсер

могао да приступи њиховим вредностима. Лабеле се додељују само за оне симболе који се користе у акцији за грађење стабла, и то тако што се назив лабеле формира од првог слова симбола и броја који је јединствен на нивоу смене. Постојеће и новогенерисане лабеле се затим користе за креирање акције за грађење новог чвора апстрактног синтаксног стабла.

Ова акција се додаје на самом крају смене и састоји се само од доделе вредности променљивој RESULT. Вредност која се додељује јесте референца на новокреирани објекат типа који одговара типу, односно називу дате смене, са конструктором који прихвата онолико аргумената колико има симбола у смени. Ако се при раду са парсером користе класе генерисане из спецификације коришћењем овог додатка, тада број и тип аргумената конструктора одговарају броју и типу поља класе која је генерисана на основу дефиниције дате смене.

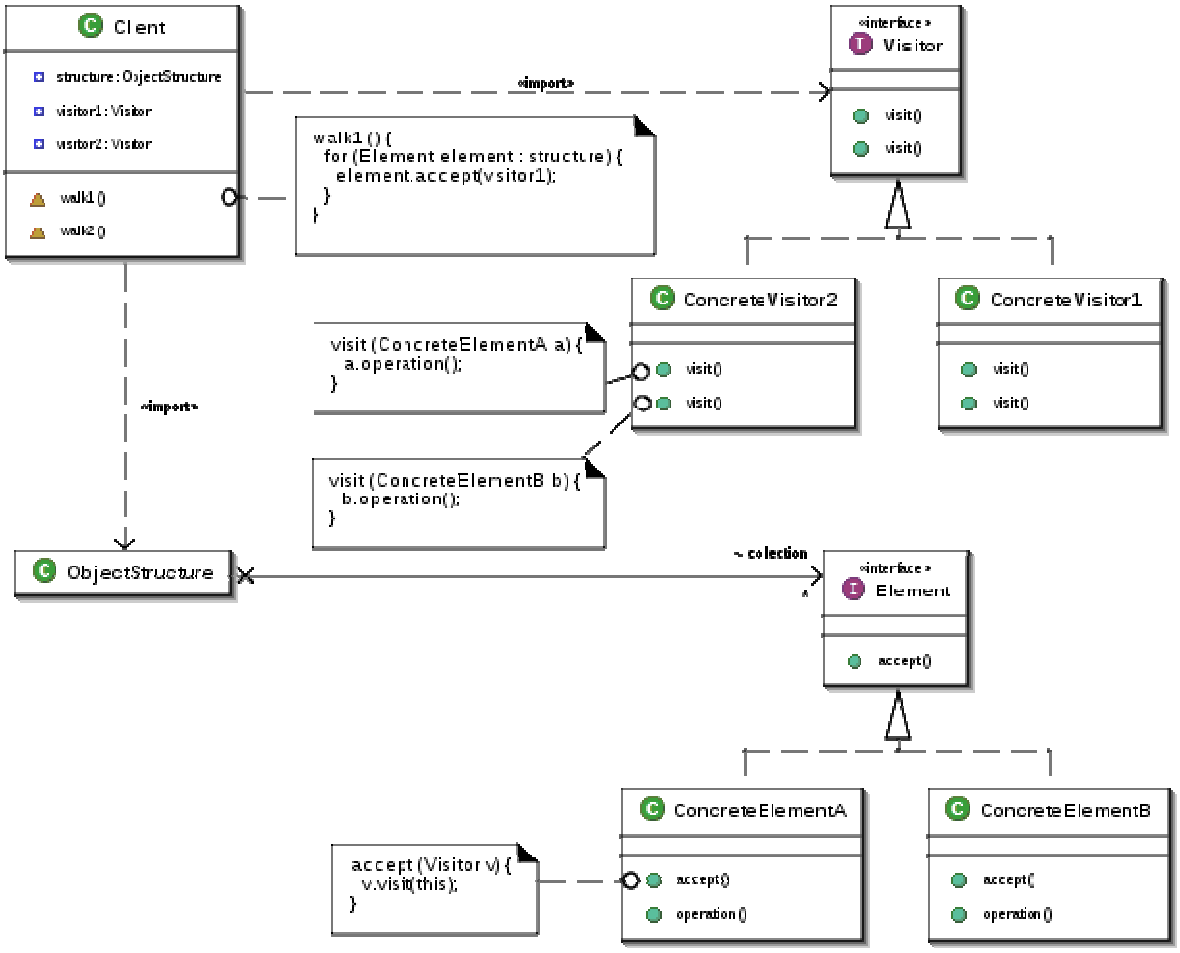
По додавању набројаних елемената у текст спецификације неке граматике, добија се улазни фајл из кога се генерише парсер који има могућност рада са апстрактним синтаксним стаблом. Вредност коју парсер враћа из своје методе *parse* је референца на чвор који представља корен апстрактног синтаксног стабла.

4. Преглед имплементације и коришћени пројектни узорци

За реализацију програмског преводиоца који користи представљање међукода у форми апстрактног синтаксног стабла најпогодније је коришћење и имплементација пројектног узорка Посетилац (Visitor). Овај узорак омогућава једноставно додавање функционалности класама које представљају чворове апстрактног синтаксног стабла без потребе за мењањем њиховог интерфејса. На пример, могуће је потпуно одвојити функционалност синтаксне и семантичке провере исправности кода и генерисања кода или његове оптимизације. Ово решење одваја имплементацију појединачних класа и имплементацију сложенијих функционалности и омогућава да се напише доста прегледнији код који је самим тим и лакши за одржавање и разумевање.

Пројектни узорак Посетилац се имплементира тако што се методе које обављају исту функционалност, само за различите чворове, групишу у посебне класе које називамо посетиоцима. Свака класа која репрезентује неки чвор стабла има методу за прихватање посете, и када се та метода позове, она шаље захтев објекту посетиоца за обављање неке радње над тим чвором и прослеђује себе као аргумент, а посетилац сада извршава жељену операцију [5]. Тиме се постиже да се сада све методе које рецимо проверавају синтаксну исправност за сваки тип чвора који се може наћи у стаблу нађу на једном месту, а такође и све променљиве које могу олакшати или уопште омогућити жељену акцију. Различитим класама посетилаца које све имплементирају заједнички интерфејс постиже се да интерфејс класе чвора није потребно мењати приликом додавања новог посетиоца.

Коришћењем овог пројектног узорка дефинишу се две одвојене хијерархије класа: једна која представља све елементе над којима се врши обрада (хијерархија чворова стабла) и друге која представља посетиоце који дефинишу све могуће операције над елементима. Све док се не мења граматика датог језика, додавање нове функционалности у компајлер се своди на проширивање друге хијерархије новим класама које имплементирају основни интерфејс Посетиоца.



Слика 1 - УМЛ дијаграм пројектног узорка Посетилац

Подршка за имплементацију пројеткног узорка Посетилац је у додатку за генерисање класа реализована тако што алат генерише основни интерфејс *Visitor* који има онолико различитих метода *visit()* колико има различитих типова чворова генерисаних на основу дате граматике. Сваки објекат неког од чворова има методу *accept()* којом се прихвата посетилац и покреће обрада над датим чвором. Такође, додате су и методе које омогућавају обилазак стабла одозго надоле или одоздо нагоре. Ове методе поред тога што позивају методу за прихватање посетиоца, позивају одговарајуће методе својих синова пре или после прихватања (рекурзивно), у зависности од жељеног начина обиласка.

Ако у неким случајевима није потребно имплементирати неку функционалност на свим чворовима који се могу наћи у стаблу, уместо имплементирања основног интерфејса *Visitor* могуће је имплементирати класу *VisitorAdaptor* која има имплементиране све методе са празним телом, па онда редефинисати само оне које су заиста неопходне.

5. Упутство за употребу алата

Проширење за аутоматско генерисање класа и градњу апстракног синтаксног стабла *ast extension* се користи у оквиру CUP генератора парсера. Приликом покретања .jar фајла који представља CUP, за коришћење додатка је потребно само навести параметре командне линије као аргументе програма. Потребно је навести аргумент *–ast imepaketa* за статичко генерисање класа неопходних за градњу апстрактног синтаксног стабла. Тада се креира нови директоријум са називом *imepaketa* и у њега се смештају све класе које додатак генерише, а генерисане класе тада припадају поменутом пакету. Други аргумент који се може употребити је аргумент *–buildtree* ако се жели да генерисани парсер има могућност динамичког генерисања стабла. Ако се наведе овај аргумент, креира се нови фајл са спецификацијом граматике, са додатим акцијама за креирање стабла, а стандардни улаз се преусмерава са оригиналног на модификовани фајл (додаје се суфикс _astbuild на име оригиналног фајла). Ова два мода коришћења додатка су међусобно независни, па је могуће користити их засебно. Наравно, могуће је покренути CUP генератор парсера и ако се не жели користити додатак, тада се само изостави опција из командне линије, тј. покрене се програм као у верзији која нема овај додатак.

5.1 Аутоматско генерисање класа

Граматика самих .cup фајлова спецификације је проширена могућношћу да се свакој продукцији додели име које ће при генерисању класа постати и име сваке од новогенерисаних класа. Име се може доделити на почетку продукције, окружено паром заграда. Ово је приказано на примеру дефиниције смене за нетерминал *program* у спецификацији граматике микројаве [6]:

```
   program ::= (ProgramDM) CLASS IDENT:name declaration_list LPAREN
method_declaration_list RPAREN
      | (ProgramM) CLASS IDENT:name LPAREN method_declaration_list RPAREN
      | (ProgramE) CLASS IDENT:name LPAREN RPAREN
      | (ProgramD) CLASS IDENT:name declaration_list LPAREN RPAREN;
```

Из овог примера се јасно може видети зашто је погодно користити могућност за директно именовање продукција у спецификацији граматике. Продукције које одговарају нетерминалу *program* се могу разликовати и без детаљнијег разматрања њихове структуре, јер се могу описати именом. Овде прва продукција има и листу декларација и листу декларација метода, па зато у њеном имену стоји суфикс DM. Трећа продукција нема ниједан од ова два елемента, па у њеном имену стоји E, скраћено од *empty*, односно ознака да продукција не садржи елементе, итд. Ово још више помаже бољем разумевању кода и прегледности спецификације граматике. Ако

се нека од продукција не именује биће јој аутоматски додељен тип са називом изведеним из назива типа нетерминала са леве стране продукције.

Приликом дефиниције нетерминалних симбола у спецификацији граматике сада је могуће и не навести тип за нетерминалне симболе који нису простог типа. Алат препознаје да се ради о нетерминалима који треба да буду представљени базним класама и аутоматски им додељује тип. Назив типа нетерминала се добија тако што се узме назив нетерминала са првим словом претвореним у еквивалентно велико слово. У претходно изложеном примеру, ако је нетерминал *program* декларисан као

```
nonterminal program;
```

алат ће му доделити тип *Program* а све класе које представљају продукције тог нетерминала ће бити изведене из класе *Program*. За нетерминале који су простих типова мора се експлицитно навести тип како би алат знао да од њих не треба да прави базне класе. За терминалне симболе за које је потребно да се нађу у стаблу неопходно је да буде дефинисан тип, јер ће на тај начин алат препознати да су они потребни да би се очувало исправно значење програма (имена променљивих, нумеричке константе итд).

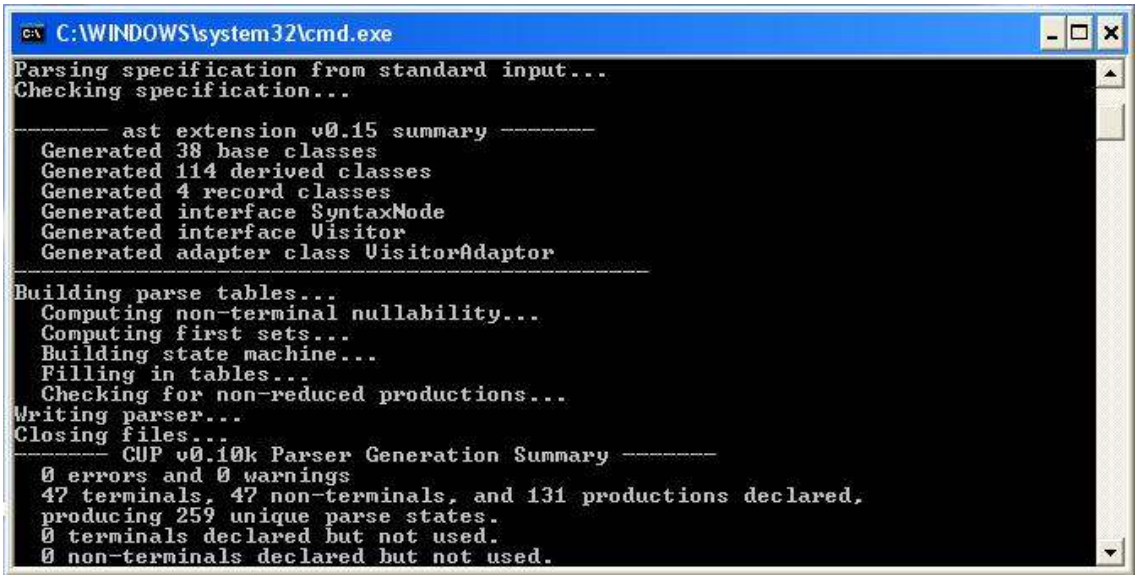
5.2 Додавање акција за градњу стабла

Ако се жели користити могућност генерисања новог фајла са спецификацијом граматике са додатим акцијама за грађење апстрактног синтаксног стабла на крају сваке продукције, потребно је обратити пажњу на следећа правила:

- При декларацији нетерминалних симбола није неопходно наводити тип. Ако се тип не наведе, парсер ће у модификованом тексту поставити одговарајући тип нетерминала.
- Ако се терминалном симболу не наведе тип, сматра се да он не треба да учествује у градњи стабла, па неће ни бити наведен у акцијама оних смена где се појављује. Сви терминали којима је при декларацији наведен тип учествују у градњи стабла.
- Ако је неки симбол, терминални или нетерминални, који треба да учествује у градњи стабла означен лабелом у телу продукције, он ће тим именом бити пресликан у тело акције. Ако не, биће му аутоматски генерисано име.
- Ако се нека смена у продукцији експлицитно не именује, за њено име ће бити употребљено исто име које би било коришћено при генерисању класе за ту смену, изведено из назива нетерминала са леве стране продукције.

Пример изгледа преуређеног текста спецификације граматике за пример дефиниције продукције нетерминала *program* наведене раније у тексту:

```
   program ::= (ProgramDM) CLASS IDENT:name declaration_list:d1 LPAREN
method_declaration_list:m2 RPAREN {: RESULT=new ProgramDM(name, d1, m2); :}
      | (ProgramM) CLASS IDENT:name LPAREN method_declaration_list:m1
RPAREN {: RESULT=new ProgramM(name, m1); :}
      | (ProgramE) CLASS IDENT:name LPAREN RPAREN
{: RESULT=new ProgramE(name); :}
      | (ProgramD) CLASS IDENT:name declaration_list:d1 LPAREN RPAREN
{: RESULT=new ProgramD(name, d1); :};
```



Слика 2 - Изглед екрана по покретању додатка

6. Поређење са постојећим решењима

На интернету се може наћи велики број алата који могу да се користе као генератори лексичких анализатора и парсера, за различите програмске језике, као и додатака за њих. Неки од њих су креирани са циљем да се подржи и рад са апстрактним синтаксним стаблима при изради преводилаца, један од њих је classgen.

Алат classgen је намењен генерисању класа у програмском језику Јава. Овај алат се може користити у спрези са CUP генератором парсера, а погодно је и то што деле сличну синтаксну спецификационих фајлова. Он је и послужио као директан модел по узору на који је имплементиран додатак ast extension па зато класе које се генеришу овим додатком изгледају потпуно идентично као класе које генерише classgen, наравно ако се користе одговарајуће идентичне, односно прилагођене спецификације и именовање поља класа.

Пример једног дела спецификације classgen хијерархије класа на претходно приказаној смени нетерминала *program* је

```
   program ::= {programDM} String:name declaration_list
method_declaration_list
      | {programM} String:name method_declaration_list
      | {program} String:name
      | {programD} String:name declaration_list
```

Сличност са CUP спецификацијом је очигледна, једино се овде не појављују терминални симболи који не учествују у саставу стабла. Ако се жели тип поља који није добијен од његовог имена, неопходно је то и навести (овде пример типа String за терминал name), док се у CUP спецификацији то имплицитно зна на основу дефинисања типа неког симбола при његовој декларацији. Идеја са именовањем продукција је пресликана из classgen спецификације и додата у граматику CUP фајлова. На основу овог сегмента спецификације ће classgen генерисати базну апстрактну класу Program и четири њене потклase са одговарајућим именима и пољима.

7. Закључак

Додатак за рад са апстрактним синтаксним стаблима *ast extension* поједностављује рад са апстрактним синтаксним стаблима и генератором парсера CUP и омогућава програмерима да се потпуно посвете изради класа које имплементирају потребне функционалности над стаблом, без потребе за коришћењем екстерних алата.

Поред тога, додатак се може, уз одређене модификације, користити и самостално за генерисање произвољне структуре класа. Једна од идеја за наставак развоја алата је да се омогући параметризовање метода генерисаних класа, додавање интерфејса који се имплементирају итд. Ово се може реализовати нпр. проширењем основне CUP граматике или екстерним конфигурационим фајлом. Прво решење доноси већу флексибилност јер се може радити на нивоу појединачних смена, као што је учињено овде приликом додавања могућности именовања смена, док је предност другог решења што је независно од основног алата.

Један од недостатака додатка може да представља то што није могуће прецизније дефинисати улогу појединачних терминалних и нетерминалних симбола у генерисаним класама и акцијама, већ се за то додатак ослања само на препознавање типова при декларисању истих. То је последице коришћења CUP граматике која није дизајнирана са тим у виду, па се и не може се очекивати да одговори посебним, накнадно постављеним захтевима.

На крају треба поменути и позитивне утиске при раду са алатом отвореног кода (open source) и предности таквог приступа. Поред тога што омогућава упознавање неких метода и решења на примерима имплементације, он омогућава програмерима да мењају и побољшавају постојећи код у складу са својим потребама или новим идејама.

8. Литература

[1] CUP генератор парсера, http://www2.cs.tum.edu/projects/cup/

[2] Alfred Aho, Monica Lam, Ravi Sethi, Jeffrey Ullman, "*Compilers Principles, Techniques and Tools"*, 2006, Addison-Wesley

[3] Енглеска Википедија, "*LALR parser"*, 2009

[4] Алат classgen, http://classgen.sourceforge.net/

[5] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, "*Design Patterns: Elements of Reusable Object-Oriented Software"*, 1994, Addison-Wesley

[6] Спецификација језика МикроЈава, http://ir4pp1.etf.rs/Predavanja/mikrojava.pdf

Додатак А – Пример

А.1 Граматика за једноставне аритметичке изразе

```
package arithmetic;

terminal int NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;

nonterminal expr, term, factor;
nonterminal String addop, mulop;

expr ::= (SingleExpr) term
   | (Addition) expr addop term;

term ::= (SingleTerm) factor
   | (Multiplication) term mulop factor;

factor ::= (Constant) NUMBER:val;

addop ::= PLUS {: RESULT="add"; :} | MINUS {: RESULT="sub"; :};

mulop ::= STAR {: RESULT="mul"; :} | DIVIDE {: RESULT="div"; :};
```

А.2 Граматика са додатим акцијама за грађење стабла

```
package arithmetic;

terminal int NUMBER;
terminal PLUS, MINUS, STAR, DIVIDE;

nonterminal Expr expr;
nonterminal Term term;
nonterminal Factor factor;
nonterminal String addop, mulop;

expr ::= (SingleExpr) term:t1 {: RESULT=new SingleExpr(t1); :}
   | (Addition) expr:e1 addop:a2 term:t3 {: RESULT=new Addition(e1, a2,
t3); :};

term ::= (SingleTerm) factor:f1 {: RESULT=new SingleTerm(f1); :}
   | (Multiplication) term:t1 mulop:m2 factor:f3 {: RESULT=new
Multiplication(t1, m2, f3); :};

factor ::= (Constant) NUMBER:val {: RESULT=new Constant(val); :};

addop ::= PLUS {: RESULT="add"; :} | MINUS {: RESULT="sub"; :};

mulop ::= STAR {: RESULT="mul"; :} | DIVIDE {: RESULT="div"; :};
```

A.3 Пример једне генерисане класе

```
package arithmetic;

public class Addition extends Expr {

    private Expr expr;
    private String addop;
    private Term term;

    public Addition (Expr expr, String addop, Term term) {
        this.expr=expr;
        if(expr!=null) expr.setParent(this);
        this.addop=addop;
        this.term=term;
        if(term!=null) term.setParent(this);
    }

    public Expr getExpr() {
        return expr;
    }

    public void setExpr(Expr expr) {
        this.expr=expr;
    }

    public String getAddop() {
        return addop;
    }

    public void setAddop(String addop) {
        this.addop=addop;
    }

    public Term getTerm() {
        return term;
    }

    public void setTerm(Term term) {
        this.term=term;
    }

    public void accept(Visitor visitor) {
        visitor.visit(this);
    }

    public void childrenAccept(Visitor visitor) {
        if(expr!=null) expr.accept(visitor);
        if(term!=null) term.accept(visitor);
    }

    public void traverseTopDown(Visitor visitor) {
        accept(visitor);
```

```
        if(expr!=null) expr.traverseTopDown(visitor);
        if(term!=null) term.traverseTopDown(visitor);
    }

    public void traverseBottomUp(Visitor visitor) {
        if(expr!=null) expr.traverseBottomUp(visitor);
        if(term!=null) term.traverseBottomUp(visitor);
        accept(visitor);
    }

    public String toString(String tab) {
        StringBuffer buffer=new StringBuffer();
        buffer.append(tab);
        buffer.append("Addition(\n");

        if(expr!=null)
            buffer.append(expr.toString("  "+tab));
        else
            buffer.append(tab+"  null");
        buffer.append("\n");

        buffer.append(" "+tab+addop);
        buffer.append("\n");

        if(term!=null)
            buffer.append(term.toString("  "+tab));
        else
            buffer.append(tab+"  null");
        buffer.append("\n");

        buffer.append(tab);
        buffer.append(") [Addition]");
        return buffer.toString();
    }
}
```

A.4 Пример посетиоца за рачунање вредности израза

```
package arithmetic;

import java.util.ArrayList;

public class ArithmeticVisitor extends VisitorAdaptor {

   ArrayList<Integer> stack=new ArrayList<Integer>();

   public void visit(Constant constant) {
         stack.add(constant.getVal());
   }

   public void visit(Multiplication multiplication) {
         int op1=stack.remove(stack.size()-1);
         int op2=stack.remove(stack.size()-1);

         if(multiplication.getMulop().equals("mul")) {
               stack.add(op1*op2);
         } else {
               stack.add(op1/op2);
         }
   }

    public void visit(Addition addition) {
         int op1=stack.remove(stack.size()-1);
         int op2=stack.remove(stack.size()-1);

         if(addition.getAddop().equals("add")) {
               stack.add(op1+op2);
         } else {
               stack.add(op1-op2);
         }
   }

   public int getResult() {
         return stack.get(0);
   }

}
```