Универзитет у Београду
Електротехнички факултет

Употреба фазера за генерисање кода за тестирање MicroJava преводиоца

Дипломски рад

Ментор:
проф. др Драган Бојић,
редовни професор

Кандидат:
Жељко Урошевић
2020/0073

Београд, Септембар 2024.

САДРЖАЈ

1. Увод

Програмски преводилац или компајлер у генералном смислу представља софтвер који претвара код из једног програмског језика у други програмски језик. Најчешћа употреба преводилаца јесте превођење кода из вишег програмског језика у машински код који се може извршавати на датој машини, што омогућава лакше писање софтвера јер виши програмски језици користе апстракције које су ближе људском начину размишљања.

Синтакса језика је скуп правила како се речи и симболи могу комбиновати тако да се формирају исправни изрази за дати језик. Тако је и у програмским језицима где синтакса дефинише структуру кода (редослед инструкција, начин декларисања променљивих и функција,…) и она се изражава граматиком. Граматика је формални систем правила помоћу којих је дефинисана синтакса језика. Она се састоји од продукционих правила (смена) како се нетерминални симболи могу заменити терминалним симболима (конкретним елементима).

Тестирање софтвера је битан корак у процесу развоја, јер осигурава да софтвер функционише исправно. Циљ тестирања је да се открију грешке, слабости или недостаци у систему пре него што се софтвер пусти у употребу. Тестирање граматика програмских језика осигурава да дефинисана граматика тачно описује синтаксу језика за коју се пише и да подржава све валидне конструкције кода. Такође то омогућава и откривање недоследности у синтакси тако да сам писац граматике може да је коригује.

Фазинг (енгл. *fuzzing*) је техника тестирања софтвера која се користи за откривање грешака у раду тако што се генерише велики број случајно генерисаних улаза, чији формат може на неки начин бити контролисан. Сам алат који генерише случајне излазе се зове фазер (енгл. *fuzzer*).

Приликом израде пројекта из предмета Програмски преводиоци 1 студенти имплементирају 4 фазе кроз које сваки компајлер пролази: лексичка анализа, синтаксна анализа, семантичка анализа и генерисање кода. Проблем настаје када студенти заврше синтаксну анализу (напишу граматику за *MicroJava* језик која одговара спецификацији за пројекат те године) и када хоће да тестирају своју синтаксну анализу, тешко је написати програмски код који би зашао у неке специјалне случајеве које дата граматика дозвољава или не дозвољава. Самим тим фазер јесте добро решење за ово, јер се базира на случајном генерисању излаза који може испитати неке случајеве којих се студенти не би сетили.

У поглављу 2 овог документа ће бити објашњен појам фазинга, зашто је настало и зашто може бити корисно. Биће описана и синтаксна анализа, као обавезан део сваког програмског преводиоца, шта је за њу потребно и шта је њен продукт. Такође ће се описати укратко сам језик *MicroJava* и његов преводилац у смислу спецификације која је потребна за синтаксну анализу и увешће се класа *ISLaSolver* која је сам центар решења.

У трећем поглављу овај рад ће се дотаћи приступа решењу односно како искористити постојећи фазер да се генерише синтаксно исправан *MicroJava* програмски код. Објаснiће се зашто није добро да се директно користи граматика програмског језика за фазинг, већ се уводи

формат граматике апстрактно синтаксног стабла и како је имплементиран сам *MicroJava* фазер да подржава произвољну спецификацију тог програмског језика.

У поглављу 4 ће се детаљно описати како је текла сама имплементација решења, односно увести две апстракције које представљају чвор апстрактно синтаксног стабла као и дескриптор тог чвора и њихова улога. Затим ће се објаснити и сам пројектни узорак Посетилац. Након тога ће се описати како је генерисана сама граматика апстрактно синтаксног стабла на основу спецификација лексера и парсера и коришћење класе *ISLaSolver* како за фазинг тако и за увођење ограничења која омогућавају контролу излаза фазера. На крају овог поглавља ће бити дате и генералне смернице око коришћења класе *MicroJavaFuzzer*.

У поглављу 5, односно у закључку, биће направљен кратак преглед урађеног, потенцијална тачка проширења самог рада и споменуто коришћење овог рада за друге преводиоце.

2. ПРЕГЛЕД ГЛАВНИХ ПОЈМОВА

Прво треба увести сам *MicroJava* [1] програмски језик, неке делове његове спецификације, затим објаснити значај синтаксне анализе и њен производ. Након тога треба увести појам *MicroJava* компајлера, односно елементе његове спецификације која је потребна за успешну израду синтаксне анализе тог компајлера, а која се користи и за фазинг различитих *MicroJava* програма. Затим ће се увести и сам појам фазинга као и основа која се користи за фазинг, а то је класа *ISLaSolver*.

2.1 Програмски језик MicroJava

Програмски језик *MicroJava* је уведен на курсу Програмски преводиоци 1 као начин илустрације како преводиоци раде на примеру правог језика. Сам језик је сличан програмском језику *Java*, само је доста једноставнији. Сваке године студенти као део овог курса имају задатак да имплементирају компајлер за програмски језик *MicroJava* по спецификацији која је актуелна за ту годину.

Неки главни елементи овог језика који се јављају у свакој спецификацији јесу да програм почиње кључном речју *program,* иза које иде име самог програма и могу се дефинисати статичке променљиве и методе, као и класе. Главна метода се увек зове *main* и од ње почиње извршавање. Од основних типова, подржани су *int* и *char*, а спецификација по којој је рађен овај рад подржава и тип *bool*. Наслеђивање класа функционише слично као у програмском језику *Java*. Подржава се само једноструко наслеђивање и редефинисање (*overriding*) метода, али не и преклапање (*overloading*) метода.

Извршно окружење је *MicroJava* виртуелна машина (*MJVM)* која је осмишљена да личи на *Java* виртуелну машину, само је прилично упрошћена. Она представља стек машину што значи да нема регистара него се све чува на стеку израза (*expression stack*) којим се барата машинским инструкцијама. Такође постоји и програмски стек, коме се приступа преко регистара *sp* (*stack pointer*) и *fp* (*frame pointer*), а он служи за чување стварних аргумената приликом позива функција.

MJVM има више области података: стек израза, програмски стек, област глобалних података и област динамичких података (*heap*). Област глобалних података је ограничена и ту се налазе статичке променљиве. У области динамичких података се смештају низови и објекти класа конструисани оператором *new*. У *MJVM* не постоји ни деалокатор ни скупљач смећа тако да једном алоцирана меморија остаје алоцирана до краја програма.

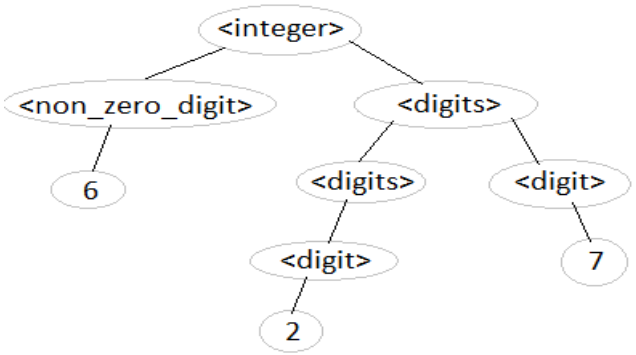
2.2 Синтаксна анализа

Синтаксна анализа је један корак у превођењу програма и користи се при конструкцији програмских преводилаца и представља проверу структурне исправности написаног програмског кода. Синтаксна анализа директно зависи од граматике, тако да да би се успешно завршила синтаксна анализа, програмски код мора да задовољава синтаксу тог језика, односно да задовољава написану граматику.

Оно што је специфично за програмске језике, јесте да се њихова синтакса исказује помоћу бесконтекстних граматика. Генерално граматика у основи има терминалне и нетерминалне симболе. Терминални симболи представљају конкретне елементе језика. Нетерминални симболи представљају апстракцију такву да се могу заменити неком листом терминалних и нетерминалних симбола. Таква замена се назива смена или продукција и један нетерминални симбол може да има више смена. У смислу бесконтекстних граматика, једна смена са леве стране мора имати један нетерминални симбол, а са десне стране произвољну листу терминалних и нетерминалних симбола, с тим да је дозвољено да се и са леве и са десне стране нађе исти нетерминални симбол. Оно што финално дефинише бесконтекстну граматику јесте почетни симбол, који представља нетерминални симбол. По правилу, нетерминални симболи се пишу унутар < >, и усвојено је да леву и десну страну смене одваја знак једнакости. Такође, више различитих продукција за једну смену је одвојено усправном цртом. Једноставан пример граматике је дефинисање целих бројева:

```
<integer> = <non_zero_digit><digits> | <digit>

<non_zero_digit> = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<digits> = <digits><digit> | <digit>

<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Апстрактно синтаксно стабло (*AST*) је структура података која репрезентује структуру програмског кода и представља резултат синтаксне анализе компајлера. Оно се састоји од синтаксних чворова, где сваки чвор представља неки терминални или нетерминални симбол. Терминални симболи су увек листови овог стабла, а деца неког нетерминалног симбола представљају једну од његових смена. Корен апстрактно синтаксног стабла јесте сам стартни симбол бесконтекстне граматике. Даље се синтаксно стабло прослеђује семантичкој анализи да се провери сама семантичка исправност написаног кода и фази генерисања кода где се сада генерише извршни код. На примеру целог броја 627, за претходну граматику *AST* би изгледало:



Слика 2.2.1. Пример апстрактно синтаксног стабла.

Када неки програмски код не задовољава граматику језика, то значи да је немогуће изградити апстрактно синтаксно стабло и самим тим није успела фаза синтаксне анализе и ту се завршава рад преводиоца. У примеру, број 05 не би могао да задовољи граматику.

2.3 MicroJava преводилац

Да би се успешно имплементирала фаза синтаксне анализе, очекује се да се напишу две датотеке које представљају спецификацију за два алата: лексер и парсер. Први алат на основу спецификације из улазног програмског кода издваја токене као лексичке целине који се прослеђују парсеру. Парсер сада на основу улазне секвенце токена гради апстрактно синтаксно стабло и проверава да ли та улазна секвенца токена одговара граматици језика (дефинисаној у спецификацији). Како рад не би постао преоптерећен описом спецификација ова два алата, само ће се објаснити елементи који су битни за прилагођавање и коришћење постојећег фазера.

Спецификација лексера се пише у датотеци са екстензијом *.lex* и служи за дефинисање токена, односно мапирање токена у низ карактера који тај токен представља. Овде се дефинишу и елементи из улазног кода које треба игнорисати (коментари) као и да знакови белине не представљају никакав токен већ их треба прескакати. Линије из датотеке које су од интереса су дефиниције токена без регуларних израза. Пример такве линије је:

```
"+"             { return new_symbol(sym.PLUS, yytext()); }
```

Оно што је овде од интереса јесте да се низ карактера између наводника третира као секвенца карактера која ће се касније препознати као токен `PLUS`. Поред тога, могу се дефинисати и регуларни изрази између наводника, који се користе, на пример, за препознавање идентификатора, бројева и слично.

Спецификација парсера се пише у датотеци са екстензијом *.cup* и овде се налазе методи који се користе за пријаву грешака у синтаксној анализи, декларације (не)терминалних симбола и дефиниција саме граматике. Оно што је коришћено у изради фазера јесу декларације терминалних симбола и смене из граматике. Пример декларације терминалних симбола је:

```
terminal RIGHT_BRACE, FOREACH, EQUALS;
```

Овде се наводи низ имена који представљају терминалне симболе у граматици, а самим тим и листове у *AST*, а ова имена одговарају токенима из спецификације лексера. Поред ових терминалних симбола, може се навести и *Java* тип тог симбола, а такви симболи су углавном у спецификацији лексера дефинисани преко регуларних израза. Пример овога је:

```
terminal String IDENT;
```

Дефинисање смена граматике има две варијанте, када нетерминални симбол има само једну смену и када има више смена битно је издвојити ова два случаја због исправног рада како преводиоца тако и фазера јер се у оба случаја узима претпоставка да је формат испоштован. Када симбол има само једну смену то изгледа овако:

```
FuncCallStart ::= (FuncCallStart) Designator LEFT_PAREN;
```

Лева страна садржи име нетерминала, у заградама **мора** бити исто име као и смена и онда долази листа терминалних и нетерминалних симбола која се завршава тачка-зарезом (;). Када симбол има више смена (у овом примеру две), оне су одвојене усправном цртом (|):

```
Expr ::= (NegativeExpr) MINUS StartExpr Term TermList
```

```
    | (PositiveExpr) StartExpr Term TermList;
```

Лева страна садржи име нетерминала, и свака смена **мора** имати посебно име наведено у заградама и та имена морају бити јединствена и разликовати се од имена нетерминала. Празна смена се означава секвенцом карактера */* epsilon */*. За опоравак од грешке приликом синтаксне анализе се може користити секвенца карактера *error*, међутим за конструкцију фазера се те смене прескачу, те се нема потребе задржавати на њима.

2.4 Појам фазера

Први пут кад се појавио појам фазинга је био 1988-е [2]. Била је јесен и олујна ноћ и тада је професор Барт Милер био повезан на универзитетски рачунар преко телефонске линије од 1200 *baud*-а. Олуја је узроковала шум на линији, што је доводило до тога да *UNIX* команде које је он уносио добијају лош улаз и због тога се дешавало да те команде престају са радом (*crash*). Толико често је долазило до тога да програми престају са радом због шума на улазу да се зачудио како је то могуће с обзиром да је то нешто што су користили сваког дана. Због овог дешавања је дошао на идеју да студентима те године да пројекат који је назвао *The Fuzz Generator* који би избацивао случајно генерисан низ карактера и доводио се на улаз разних *UNIX* команди све у покушају да их доведе до пада [3].

Дакле, из ове приче се може закључити да је техника фазинга у тестирању софтвера у ствари генерисање случајног низа карактера и преусмеравање тог низа карактера на улаз софтвера све у циљу изазивања краха. Након тога би се анализирало зашто је дошло до краха и имплементирао опоравак од неког таквог улаза. На пример, улаз генерисан на фази начин се може довести на неко поље форме неке веб странице или као неки аргумент командне линије за неки програм.

Ако би се посматрало тестирање форми веб страница, поља некад очекују улаз у одређеном формату. На пример за унос године рођења, поље форме ће очекивати само број, па нема смисла генерисати потпуно случајан низ карактера, већ некако контролисати каквог формата ће излаз фазера бити. Тај формат је најбоље дефинисати бесконтекстном граматиком.

Процес фазинга из дате граматике на неком базичном нивоу функционише тако што се почне од стартног симбола, ако има више смена на случајан начин се изабере којом сменом ће се стартни симбол заменити, проверава да ли и даље постоји неки нетерминални симбол, и ако постоји узима се први по реду и понавља се поступак рекурзивно све док не остану само терминални симболи. Када остану терминални симболи, онда је добијен излаз фазера.

2.5 Класа ISLaSolver

Доминик Стаинхофел и Андреас Зелер [7] су направили посебан језик за спецификацију који омогућава да се производи излаз са произвољним особинама без потребе за имплементацијом неких посебних провера који су назвали *ISLa* (*Input Specification Language*). Овај језик комбинује бесконтекстну граматику са ограничењима да би се изразила семантичка правила за генерисани излаз, тако да су ограничења писана у овом језику. *ISLa* може да се користи и као фазер и као алат за проверу да ли неки излаз испуњава дату граматику и дата ограничења.

ISLa постоји као *Python* модул који се зове *isla-solver*, и који обезбеђује дефиницију класе *ISLaSolver*. Ова класа у ствари и представља сам фазер својом методом *solve*, помоћу које се на основу задате граматике и ограничења (ова два параметра се прослеђују кроз конструктор) генерише фази излаз. Такође има могућност да провери да ли нека секвенца карактера одговара граматици (метода *check*), као и да мутира (уведе мале измене) неку постојећу секвенцу карактера тако да и даље задовољава граматику (метода *mutate*).

У овом раду ће се користити само метода *solve* јер је главни циљ генерисати синтаксно исправне *MicroJava* програме. За коришћење метода *check* и *mutate* је потребно имплементирати сам преводилац који би постојећи *MicroJava* код пребацио у репрезентацију апстрактно синтаксног стабла (која ће касније бити објашњена), и онда да се та репрезентација или провери да ли је валидна по правилима граматике и ограничењима или да се мутира.

3. ИДЕЈНО РЕШЕЊЕ

Фазер који користи граматику за генерисање случајног излаза постоји имплементиран као *fuzzingbook* [4], нестандардан модул у програмском језику *Python*. Тако да се и сама конструкција *MicroJava* фазера заснива на креирању одговарајуће граматике и прилагођавању постојећег фазера да емитује синтаксно исправан *MicroJava* програмски код. Тај постојећи фазер је у ствари инстанца класе *ISLaSolver*.

3.1 Формирање граматике

Као што је речено, прво је потребна бесконтекстна граматика на основу које ће се вршити фазинг. Прва помисао је погледати спецификацију програмског језика *MicroJava* и на основу ње написати граматику. Међутим овај приступ, иако могућ, је врло неефикасан. У случају неке јако једноставне граматике где је почетни нетерминални симбол <*expr*>:

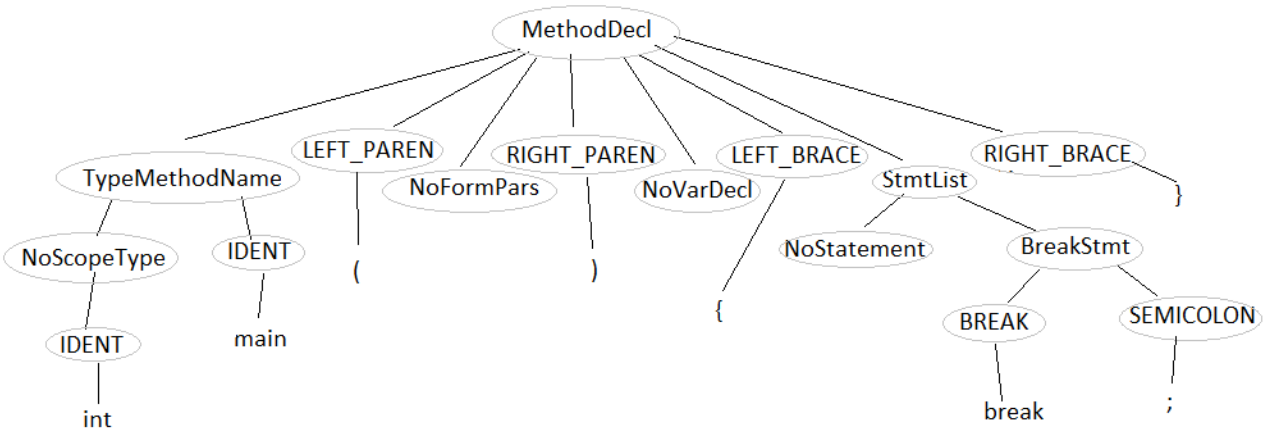
```
<expr> = <number>+<number>

<number> = <number><digit> | <digit>

<digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Нешто што одговара овој граматици је секвенца карактера 2+3, међутим секвенца карактера која садржи знаке белине (2 + 3) не одговара дефинисаној граматици. Ово би стварало јако велики проблем ако би циљ био проверити да ли неки већ постојећи код припада дефинисаној граматици или га мутирати. Решење за ово је да се између свака два (не)терминална симбола уведе произвољна секвенца знакова белине, што би преоптеретило граматику, а самим тим и успорило генерисање секвенце карактера јер би свака произвољна секвенца знакова белине морала да се развија на случајан начин.

Како је класична репрезентација кода неефикасна за фазинг, посматра се репрезентација кода преко апстрактно синтаксног стабла. Што се тиче ове репрезентације, сада треба осмислити формат секвенце карактера која ће на јединствен начин идентификовати стабло. Најједноставније је применити приступ налик *preorder* обиласку стабла. Наиме, када се посећује неки чвор, емитује се његово име, отвори се заграда и онда се посећују његова деца на исти начин, с тим да се после посете сваког детета емитује зарез, и када се посети последње дете емитује се затворена заграда. Нека је пример за ово секвенца карактера:

```
MethodDecl(TypeMethodName(NoScopeType(IDENT(K)),IDENT(x)),LEFT_PAR
EN(),NoFormPars(),RIGHT_PAREN(),NoVarDecl(),LEFT_BRACE(),StmtList(
NoStatement(),BreakStmt(BREAK(),SEMICOLON())),RIGHT_BRACE())
```

Овде се може видети да је корен стабла чвор са именом *MethodDecl* и са слике 3.1.1. јасно који чворови су му деца. Треба приметити са слике да заокружене речи представљају стварне чворове стабла, док су незаокружене речи оно што одговарајући родитељи емитују као код.



Слика 3.1.1. Пример апстрактно синтаксног стабла (*MethodDecl*).

Сада када би се обишло ово стабло у *bottom-up* маниру, и за сваки чвор при посети позвали његову методу за емитовање кода (у смислу када се посећује чвор *BREAK* емитоваће се секвенца карактера *break*), добила би се дефиниција функције *main*:

```
int main(){
      break;
}
```

На основу свега наведеног, може се написати део граматике који одговара овом стаблу, као неки пример како ће изгледати цела граматика апстрактно синтаксног стабла:

```
<TypeMethodName> = TypeMethodName(<NoScopeType>,<IDENT>)
<NoScopeType> = NoScopeType(<IDENT>)
<IDENT> = IDENT(<character_list>[1])
```

3.2 Произвољна спецификација програмског језика MicroJava

Један начин прилагођавања фазера са сада осмишљеном граматиком јесте да се узме једна спецификација програмског језика *MicroJava*, да се за њу ручно испише граматика, као и само апстрактно синтаксно стабло са све обиласцима, и да се тако генерише програмски код. Олакшица код овог приступа јесте што је сада у потпуној контроли емитовање кода, па се може направити јако фино форматиран програмски код. Једна велика мана овог приступа је сама поновна употреба, просто такав фазер ће генерисати код за само једну спецификацију програмског језика *MicroJava,* што је непрактично како се сама поставка пројекта мења на свака два рока.

Стога, овде је усвојено решење да се спецификација програмског језика *MicroJava* чита из *.lex* и *.cup* датотека, на основу њих направи граматика за *AST* и сама његова структура, и то тако да класи *ISLaSolver* на генерисање секвенце карактера, од које се прави *AST* и на основу којег се емитује *MicroJava* програмски код.

[1] Овај нетерминал је даље произвољан низ карактера где је први карактер неко слово енглеске абецеде, док остали поред слова абецеде могу бити и бројеви и доња црта.

4. Имплементациони детаљи

У овом поглављу се сада детаљније објашњава сама конструкција граматике и апстрактног синтаксног стабла, као и остали детаљи имплементације. Уводи се појам ограничења (енгл. *constraints*) која служе да се донекле контролише како ће изгледати излаз фазера.

4.1 Чвор апстрактно синтаксног стабла

Апстрактна класа која представља натклasu за све конкретне чворове апстрактно синтаксног стабла се назива *SyntaxNode* и налази се у модулу *MicroJavaAST* дефинисана у датотеци *SyntaxNode.py* и њен интерфејс је:

```
class SyntaxNode(ABC):
     def accept(self, visitor)
          """
          Call the visit method of this node.
          """

     def emit_code(self, tab)
          """
          Returns a string that is part of the code this node
          represents.
          """

     def traverse_bottom_up(self, visitor)
          """
          Traverse the tree that this node represents in a bottom
          up manner.
          """
```

Листинг 4.1.1. Класа апстрактно синтаксног чвора

Методе *accept* и *traverse_bottom_up* представљају део пројектног узорка посетилац (касније ће бити више речи о томе како је он имплементиран). Метода *traverse_bottom_up* прво позива ту методу над децом овог чвора, па затим тек позива методу *accept* над собом, која у ствари представља акцију посећивања чвора. Деца чвора се могу чувати на два начина, као низ објеката типа *SyntaxNode* или као атрибути објекта чвора. Одабрана је друга варијанта јер приликом инспекције конкретних класа чворова, лакше је уочити која су тачно деца одређеног чвора.

4.2 Дескриптор чвора апстрактно синтаксног стабла

У *config* фолдеру пројекта, налазе се 4 датотеке које имају сврху да корисник дефинише како би хтео да форматира излазни *MicroJava* програмски код. Наравно, ово је доста сиромашно форматирање, али је више ту како не би цео програмски код био емитован као једна линија, већ да донекле буде читљив. Ако се баци поглед на формат линије *.lex* датотеке који је од интереса, ту је дефинисано ком делу програмског кода (секвенца карактера између

наводника) се придружује одговарајуће име токена. Име токена ће представљати име класе синтаксног чвора који ће бити изведен из класе *SyntaxNode*. Оно што ће та класа емитовати као код из методе *emit_code* ће бити дефинисано секвенцом карактера између наводника, као и типовима којима овај синтаксни чвор припада. Баш ове 4 датотеке дефинишу те типове:

- Датотека *newline_adders*: после кода за ове чворове додаје се знак за нови ред.
- Датотека *space_adders*: после кода за ове чворове додаје се један бланко знак.
- Датотека *tab_adders*: после кода за ове чворове следи нови ред, а затим се интерно повећава ниво идентације кода за 1.
- Датотека *tab_removers*: прво се интерно смањи ниво идентације за 1, па се емитује знак за нови ред и онда код за тај чвор.

Чвор се дефинише да припада одређеном типу тако што се у посебном реду у одговарајућој датотеци остави код који он емитује (секвенца карактера између наводника). Овакав тип чворова се може назвати терминалним јер се у декларацији *.cup* они наводе као терминали.

Други тип чворова који се декларишу као терминали су чворови који уз себе имају придружен неки *Java* тип. Они су у коду названи као стандардни терминали, зато што представљају неке стандардне типове из програмског језика *Java*. Они немају фиксан унапред дефинисан код који емитују, већ тај код испуњава неки регуларни израз који је дефинисан тим *Java* типом. На пример ако се посматра декларација:

```
terminal Boolean BOOL;
```

Генератор граматике (о којем ће касније бити више речи) ће у граматику додати овакве смене за ову декларацију:

```
<BOOL> = BOOL(<_boolean>);

<_boolean> = true | false;
```

Остали *Java* типови који су подржани су *Integer*, *Character*, и *String*. Посебно треба обратити пажњу на *String* тип, јер ако се у имену тог терминала нађе нешто налик на *ident*, генератор граматике ће генерисати смене које одговарају регуларном изразу за идентификаторе, а не за ниске карактера[2]. Оно што је специфично за стандардне терминале јесте да они подразумевано припадају типу *newline_adders*.

Сама класа *SyntaxNodeDescriptor* се налази у датотеци *SyntaxNodeDescriptor.py* у модулу *MicroJavaAST* и садржи информације које су потребне како би се дефинисала класа чвора стабла коју овај дескриптор описује. Параметри конструктора ове класе су име чвора који овај дескриптор описује, име наткласе (*SyntaxNode* или нека потклаcа класе *SyntaxNode*), код који ће тај чвор да емитује и листа имена чворова који представљају параметре конструктора тог чвора и самим тим децу тог чвора.

```
class SyntaxNodeDescriptor():
    def __init__(self, class_name, base_class_name, code,
constructor_params)
```

[2] Регуларни израз за идентификаторе је [a-zA-Z][a-zA-Z0-9_]*, док је за *string*-ове ".*"

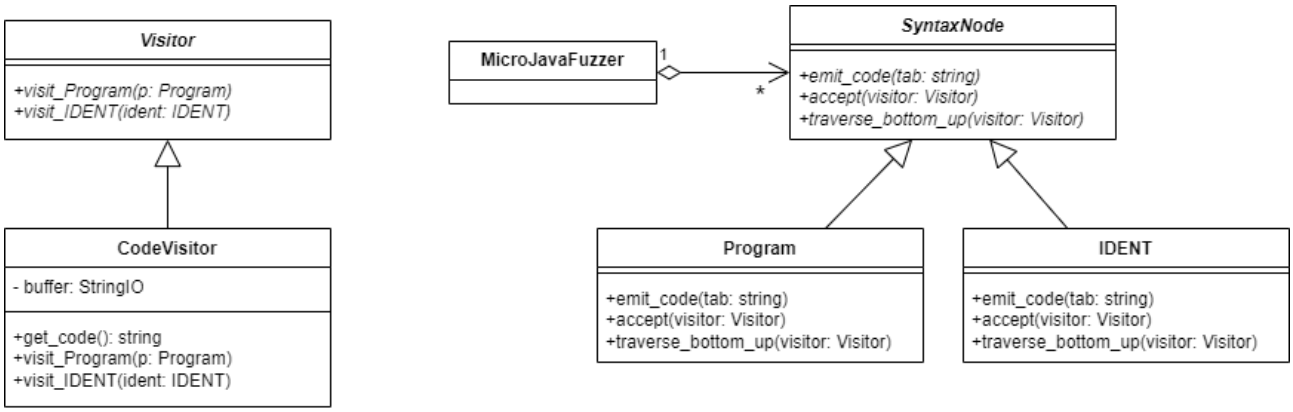
```
    def emit_class(self, space_adders, newline_adders)
        """
        Returns a string that is the definition of the class
described with this object.
        """
    def is_standard_terminal(self)
        """
        Determines whether the SyntaxNode described by this object
is a standard terminal. A standard terminal is a SyntaxNode which
is emited like an arbitrary series of characters determined by its
field _value.
        """
    def get_in_degree(self)
        """
        Determines the number of unique SyntaxNodes this object
depends on (used for determining the order SyntaxNode classes are
written to the file so every SyntaxNode class has the SyntaxNodes
that it depends on defined).
        """

    def is_dependent(self, class_name)
        """
        Determines whether the object is dependent on the
SyntaxNode class name passed as the argument.
        """
```

Листинг 4.2.1. Класа дескриптора апстрактно синтаксног чвора

Употреба метода ових класа ће се описати у одељку о генератору граматике.

4.3 Пројектни узорак Посетилац

Како је већ описано, после генерисања секвенце карактера која представља *AST*, потребно је изградити то стабло и затим га обићи у *bottom-up* редоследу. Сваки чвор има имплементиран начин на који ће се обићи његова деца (*traverse_bottom_up*) и треба само да води рачуна какав код он емитује. Ако би се на чвору оставило и само формирање и форматирање излазног програмског кода, то би било много одговорности остављено на једну класу. Како је циљ систематично обићи стабло, а да се операције при посећивању чворова одвоје од самих чворова, идеалан пројектни узорак је посетилац [5].



Слика 4.3.1. Пројектни узорак посетилац.

Оно што није потпуно објектно-оријентисано у примени овог узорка јесте то што посетиоци имају посебно име методе *visit* за сваки тип чвора. Ово је директна последица тога што програмски језик *Python* не подржава *name overloading*, па уместо што ће сваки чвор у својој *accept* методи позивати *visitor.visit(self),* мора да позове нпр. *visitor.visit_Program(self).*

Сама примена овог узорка је сада једноставна. Класа *MicroJavaFuzzer* у себи врши фазинг секвенце карактера која представља *AST* како је већ описано, затим се на основу те секвенце конструише и само стабло, после чега постоји референца на корен тог стабла *root.* Затим се направи објекат *visitor* класе *CodeVisitor* и над кореном се позове метода *root.traverse_bottom_up(visitor)*. Већ је објашњено у одељку о синтаксном чвору како се понашају методе *traverse_bottom_up* и *accept* тако да је сада време објаснити *visit* методу класе *CodeVisitor*. Оно што је сада на одговорности те класе јесте формирање излазног програмског кода. У основи, свака *visit* метода ће за прослеђен чвор као аргумент емитовати његов код у *buffer*. Оно где се прави разлика јесте код чворова који имају тип *tab_adders* или *tab_removers* где је на одговорности ове класе да интерно регулише идентацију повећавајући, односно смањујући њен ниво када треба преко интерне променљиве *tab* која представља одговарајућ број табулатора и онда тај низ карактера прослеђивати сваком позиву *emit_code* где чвор користи то само ако додаје нови ред у код.

4.4 Генератор граматике

Главна логика генерисања граматике, синтаксних чворова као и класе посетиоца се налази у класи *SyntaxNodeGenerator* лоцираној у модулу *MicroJavaAST* у датотеци *SyntaxNodeGenerator.py*. Може се приметити да она генерише три различите ствари и да се то може чинити као да је доста одговорности остављено на ову класу, међутим сва ова генерисања су толико уско повезана да није имало смисла одвајати их на више класа. Сам интерфејс ове класе је јако једноставан.

```
class SyntaxNodeGenerator():

    def __init__(self, lex_file: str, cup_file: str):

    def generate(self) -> Grammar:
        """
```

```
        Read the .lex and .cup files, form the grammar, SyntaxNode
classes and Visitors.
        """
```

Листинг 4.4.1. Класа генератора граматике

Интерно, *.lex* и *.cup* датотеке се налазе *input* фолдеру корена пројекта. Тако да главна *Python* скрипта узима те путање за ове датотеке, и направи објекат генератора граматике, генерише граматику (и у позадини синтаксне чворове и посетиоца) и ту граматику проследи класи *MicroJavaFuzzer*.

У самом конструктору генератора се читају датотеке из *config* фолдера и интерно се формира који токени припадају ком типу (*newline_adder, space_adder, tab_adder* или *tab_remover).*

Метода *generate* прво чита спецификацију лексера посматрајући само оне линије које не дефинишу токене преко регуларних израза и на основу ње гради дескрипторе синтаксних чворова, где се као код емитује оно што је написано између знака наводника, име синтаксног чвора ће бити име токена, базна класа за овакве синтаксне чворове ће бити *SyntaxNode* и ови чворови немају децу, тако да је *constructor_params* празна листа.

Затим се чита спецификација парсера из *.cup* датотеке. Прво се читају линије које наводе терминале који немају типове и ове линије се само користе као провера да ли су терминали који су ту декларисани стварно дефинисани у спецификацији лексера. Затим се читају терминали који имају придружене уз себе *Java* типове. Као што је већ наведено, за сада су подржана само 4 типа: *Integer*, *Character*, *Boolean*, и *String*. За тип *Boolean* је већ илустровано шта ће се додати у граматику. За остале типове смене у граматици би изгледале:

- ```
  terminal Integer NUMBER;
  <NUMBER> = NUMBER(<_integer>);
  <_integer> = <_non_zero_digit><_digit>*[3];
  <_digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
  <_non_zero_digit> = 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9;
  ```
- ```
  terminal String IDENT;
  <IDENT> = IDENT(<_ident>);
  <_ident> = <_start_ident_char><_ident_char>*;
  <_start_ident_char> = a | b | ... | z | A | B | ... | Z;
  <_ident_char> = a | b | ... | z | A | B | ... | Z | _ |
  0 | 1 .. | 9;
  ```
- ```
  terminal String STRING;
  <STRING> = STRING("<_string>");
  <_string> = <_character>*;
  <_character> = // садржи сваки карактер //;
  ```
- ```
  terminal Character CHAR;
  <CHAR> = CHAR('<_character>');
  <_character> = // садржи сваки карактер //;
  ```

[3] Сам модул *fuzzingbook* дозвољава да се користе звездица, плус или упитник као индикатори понављања и сходно томе има уграђену функцију која од овакве граматике конструише *EBNF* граматику коју даље користи *ISLaSolver*.

Оно што је битно напоменути јесте да имена ових смена (NUMBER, IDENT, STRING, CHAR) дефинише корисник у својој спецификацији. Такође, треба опет напоменути да ће генератор граматике разликовати смене за тип *String* ако у имену тог стандардног терминала стоји нешто налик *ident*. За типове осим ова 4 подржана ће се исписати порука кориснику како тај тип није подржан.

Што се тиче прављења дескриптора за синтаксне чворове који су стандардни терминали, име класе синтаксног чвора је онакво какво је корисник дефинисао, базна класа је *SyntaxNode*, а параметар конструктора и код који ова класа емитује је специфична вредност *_value*. Ова вредност не представља ниједан тип синтаксног чвора, већ служи за чување информације о томе какав код овај синтаксни чвор треба да емитује. Враћајући се на пример апстрактно синтаксног стабла за *MethodDecl* који је наведен, може се видети конструкција синтаксног чвора *IDENT* који представља стандардни терминал дефинисан као горе. У овом примеру постоје две инстанце оваквог чвора, с тим да један чвор као *_value* узима низ карактера *int*, а други узима низ карактера *main*.

```
def emit_code(self, tab):
     return str(self._value) + " "[4]
```

Листинг 4.4.2. Метода за емитовање кода за стандардне терминале

Последња ствар коју треба прочитати из спецификације парсера јесу саме продукције, и од њих формирати дескрипторе синтаксних чворова. Овде се сада разликују два случаја: када нетерминални симбол има једну смену и када има више од једне смене.

Ако се прво посматра нетерминални симбол са једном сменом, ту је лева страна и оно што се налази у заградама исто, па самим тим се прави дескриптор синтаксног чвора коме је име класе тај низ карактера са леве стране, базна класа је *SyntaxNode*, параметри конструктора синтаксног чвора су сви наведени терминали и нетерминали са десне стране смене. Што се тиче кода који емитује овај синтаксни чвор, оставља се као празно зато што само терминални чворови емитују код директно, док нетерминални чворови емитују код индиректно, посећујући своје стабло чији су листови сами терминални симболи.

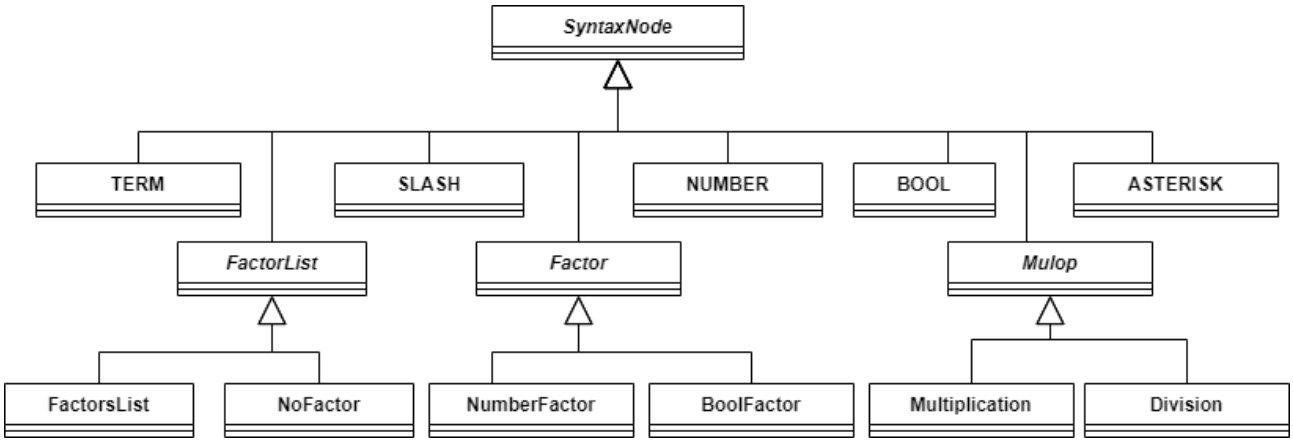
Што се тиче нетерминалних симбола који имају више од једне смене, ту је битно да оно што се налази са леве стране и оно што се налази у загради било које од ових смена не буде исто због самог формирања дескриптора. Када се дође до тренутка конструисања дескриптора, ту се види да оно што се налази са леве стране и оно што се налази у заградама није исто, па се прави хијерархија класа таква да оно што се налази са леве стране представља име базног синтаксног чвора за ту смену, а оно што се налази у заградама представља име изведеног синтаксног чвора. Базни синтаксни чвор ће бити апстрактна класа. Проверава се да ли тај дескриптор базног синтаксног чвора постоји, и ако не постоји прави се тако што је сада њему базна класа *SyntaxNode*, име класе је име базног синтаксног чвора, а код и параметри конструктора су празни. Затим се прави дескриптор изведеног синтаксног чвора коме је име класе име изведеног синтаксног чвора, име базне класе је сада име базног синтаксног чвора, а параметри конструктора су сви наведени терминали и нетерминали у тој смени. Што се тиче кода, иста дискусија важи као и горе.

[4] Приметити додавање бланко знака. Као што је већ речено, сви стандардни терминали су подразумевано типа *space_adders*.

На примеру се лакше илуструје шта се додаје у граматику, а и сама хијерархија синтаксних чворова ће бити јаснија. Посматраће се само део спецификације парсера:

```
Term ::= (Term) Factor FactorList;

FactorList ::= (FactorsList) FactorList Mulop Factor
               | (NoFactor) /* epsilon */;

Factor ::= (NumberFactor) NUMBER
               | (BoolFactor) BOOL;

Mulop ::= (Multiplication) ASTERISK
          | (Division) SLASH;
```

Пратећи поступак описан горе како се формирају дескриптори синтаксних чворова, може се закључити како ће изгледати сама хијерархија класа синтаксних чворова:



Слика 4.4.1. Хијерархија класа синтаксних чворова.

Надовезујући се на опис секвенце карактера која представља *AST*, речено је да када се посећује неки чвор, емитује се његово име. Битан детаљ овде је да ће се посећивати само конкретни синтаксни чворови, тако да никад у том испису апстрактно синтаксног стабла неће бити имена апстрактних класа. Ако се посматрају нетерминални симболи који имају једну смену, ту нема никаквих апстрактних класа, тако да је циљ директно позвати конструктор синтаксног чвора чије је име исто као име смене. Што се тиче параметара конструктора, они се узимају тако директно, редом из смене, и било да представљају терминалне или нетерминалне чворове, убацују се између знакова < и > да се означе као нетерминални симболи у граматици апстрактно синтаксног стабла. На пример када се додаје синтаксни чвор *Term* у граматику апстрактно синтаксног стабла, та смена ће изгледати овако:

```
<Term> = Term(<Factor>,<FactorList>);
```

Даље, када се посматрају нетерминални симболи који имају више смена, речено је да је име класе апстрактног синтаксног чвора у ствари нетерминални симбол из спецификације. Ако се сада погледају смене из спецификације заједно са сменом <*Term*>, види се да баш та имена улазе као имена нетерминалних симбола у граматику апстрактно синтаксног стабла.

Сходно томе, логично је да ће остале смене у граматици апстрактно синтаксног стабла изгледати овако:

```
<FactorList> = FactorsList(<FactorList>,<Mulop>,<Factor>) |
NoFactor();

<Factor> = NumberFactor(<NUMBER>) | BoolFactor(<BOOL>);

<Mulop> = Multiplication(<ASTERISK>) | Division(<SLASH>);
```

Дакле, у нетерминалним симболима који имају више смена, када се формира граматика за *AST*, име апстрактног синтаксног чвора представља име нетерминалног симбола у тој граматици, тај нетерминални симбол ће имати више смена и у свакој смени ће се звати конструктор класе чије је име у ствари име изведене класе синтаксног чвора. Ако се претпостави да је *Term* почетни симбол, пример секвенце карактера која репрезентује *AST* и која одговара дефинисаној граматици за апстрактно синтаксно стабло је:

```
Term(NumberFactor(NUMBER(7)),FactorsList(NoFactor(),Multiplication
(ASTERISK()),NumberFactor(NUMBER(5671948254031))))
```

Када би се конструисало ово стабло и обишло посетиоцем *CodeVisitor* емитовао би се код:

```
7 * 561948254031.
```

Следећа функционалност генератора граматике је генерисање класа синтаксних чворова на основу изграђених дескриптора. За ту и неке друге помоћне функције постоји датотека *ast_util.py* из *MicroJavaAST* модула.

Сама та функција пише све класе синтаксних чворова у фајл *MicroJavaAST/ast_nodes.py*, једну за другом. Одустало се од решења да свака класа синтаксног чвора има своју датотеку где је дефинисана због ограничења *Windows* оперативног система да третира имена фајлова на *case-insensitive* начин. На пример, дефинисан је токен *PROGRAM* и смена у спецификацији парсера:

```
Program ::= (Program) ProgramName NamespaceList GeneralDeclList
LEFT_BRACE MethodDeclList RIGHT_BRACE;
```

Од токена би требало да се направи класа са именом *PROGRAM* у датотеци *PROGRAM.py* и класа са именом *Program* у датотеци *Program.py* што оперативни систем *Windows* посматра као да је иста датотека, па ће садржај само преписати преко друге и ефективно изгубити класу једног синтаксног чвора.

Како би класе синтаксних чворова могле да се испишу једна за другом тако да је за сваку класу изнад ње дефинисана њена наткласа или типови њеног конструктора, сви дескриптори синтаксних чворова који постоје морају да се тополошки сортирају па су због тога у класи дескриптора уведене метода за рачунање улазног степена (*get_in_degree)* и метода провере да ли тај синтаксни чвор који је описан дескриптором зависи од имена класе синтаксног чвора прослеђене као параметар (*is_dependent*). Само тополошко сортирање тече на стандардан начин. За сваки дескриптор се прво израчуна улазни степен. Онда се узме неки дескриптор са улазним степеном нула, упише се методом *emit_class* у датотеку *ast_nodes.py* и

свим дескрипторима који зависе од овог дескриптора се улазни степен смањи за један. Овај поступак се понавља док се не истроше сви дескриптори (у случају да је овај граф зависности цикличан, бациће се изузетак *CyclicDescriptorDependency* дефинисан у *MicroJavaAST/exceptions.py*). Приликом генерисања синтаксних чворова, битно је да имати информацију који су чворови *space_adders,* а који *newline_adders,* да би се приликом креирања метода *emit_code* за синтаксне чворове из *space_adders* додао бланко знак, а за синтаксне чворове из *newline_adders* додао знак за нови ред и онда параметар *tab* те методе.

Последња функционалност генератора граматике јесте генерисање класа посетилаца, односно апстрактне класе *Visitor* и изведене класе из те, *CodeVisitor*. Оне ће се налазити у оквиру *MicroJavaAST.visitors* модула. Класа *Visitor* ће само дефинисати апстрактне *visit* методе за сваки синтаксни чвор за које информације добија из дескриптора синтаксних чворова.

Што се тиче класе *CodeVisitor*, за њено генерисање су потребне и информације који синтаксни чворови су *tab_adders,* а који *tab_removers* како би се интерно мењао ниво идентације у *visit* методама одговарајућих синтаксних чворова. Поред тога, уводи се и метода *get_code* која враћа резултат посећивања целог стабла, односно у овом случају *MicroJava* програмски код.

4.5 MicroJava фазер

Већ је речено да ће се класа *ISLaSolver* користити како би се генерисао фази *MicroJava* програмски код. Конструктор класе *ISLaSolver* прима као аргументе стартни симбол граматике одакле ће почети сам фазинг процес, као и граматику на основу које врши фазинг. Идеја је направити класу *MicroJavaFuzzer* чија је наткласа *ISLaSolver*, и онда дефинисати метод *fuzz* тако да враћа синтаксно исправан *MicroJava* програмски код.

Метода *fuzz* се састоји од позива *solve* методе наткласе *ISLaSolver*, чиме ће се генерисати случајна секвенца карактера која одговара граматици која је претходно описана. Затим је потребно изградити само апстрактно синтаксно стабло, што је енкапсулирано у функцији *construct_ast* из датотеке *ast_util.py*. Процес конструкције стабла је једноставан знајући каква је његова знаковна репрезентација. Уводи се помоћна структура *Info* која је редукована верзија дескриптора синтаксног чвора у смислу да има томе има само два поља: *class_name* што представља име класе синтаксног чвора који треба направити и *constructor_params* што представља листу објеката која се прослеђују као аргументи конструктора приликом прављења синтаксног чвора. Процес изградње тече тако што се читају знакови из секвенце и то чува као тренутно име класе. Када се дође до отворене заграде, тада се прави објекат класе *Info* са тренутним именом и ставља на стек, и почиње читање новог имена класе. Када се дође до затворене заграде, тада се скида објекат типа *Info* са врха стека и прави се објекат те класе. Ако постоји још неки *Info* објекат на стеку, онда се овај направљени објекат додаје у његове параметре конструктора. У супротном то значи да је направљен корен стабла што се враћа као повратна вредност ове функције. Финални корак јесте коришћење посетиоца *CodeVisitor* како би се обишло добијено стабло и тако продуковао *MicroJava* програмски код. Сам овај поступак је већ објашњен у одељку о пројектном узорку Посетилац.

Сама скрипта која генерише *MicroJava* програмски код и ставља га у датотеку *output/output.mj* је *main.py* која се покреће позивом команде *python main.py*. Успешно покретање ове скрипте је једино могуће ако су инсталиране све библиотеке из датотеке *requirements.txt*. Додатно, ова скрипта има и опциони именован аргумент командне линије

start_symbol којим се дефинише који синтаксни чвор ће представљати корен апстрактно синтаксног стабла. Подразумевано, овај аргумент узима вредност <*Program*>, тако да се препоручује приликом писања спецификације парсера да прва стартна смена има назив *Program*.

```
lex_file = # путања до .lex датотеке
cup_file = # путања до .cup датотеке
output_file = # путања до излазне датотеке
generator = SyntaxNodeGenerator(lex_file, cup_file)
ast_grammar = generator.generate()
fuzzer: MicroJavaFuzzer =
MicroJavaFuzzer(start_symbol=start_symbol, grammar=ast_grammar)
with open(output_file, "w") as file:
     file.write(fuzzer.fuzz())
```

Листинг 4.5.1. Употреба класа *SyntaxNodeGenerator* и *MicroJavaFuzzer* за генерисање *Microjava* програмског кода

Примери који су коришћени изнад за илустровање како изгледа репрезентација апстрактно синтаксног стабла су директно узети из неког покретања ове скрипте (с тим да су у примеру за *MethodDecl* вредности *IDENT* чворова промењени на смисленије вредности). Приликом покретања скрипте, неки од излаза су:

- Подразумевани стартни симбол:

```
1) program ut const h :: D J = false ;
   {
     void C () {
     }
   }
2) program t Hsf_R34D0orvB5Qn ::
   k6buFdAhycGTl8MNmV2xJUOaqtiC9YZwEjXgk ;
   {
   }
3) program J namespace H {
     {
     }
   }
   w j ;
   {
   }
```

- `start_symbol=<MethodDecl>:`

```
A g1wk5xh (oM :: DE X7o ) {
}
```

- `start_symbol=<Term>:`

```
false / '3'
```

- `start_symbol=<ClassDecl>`

```
class s {
     q7iMj4sqKZFVngtXYxIfzeNDBLk_A91dpTCWQvyHESua5PlhJoU
G02b8Rr36cmOw c [] ;
}
```

- `start_symbol=<Namespace>`

```
namespace HO {
     gzhr :: O K [] , BH8 ;
     {
     }
}
```

Овде се јасно види да генерисани код нема никаквог семантичког смисла, већ само испуњава граматику на основу које је генерисан. Такође, посматрајући прву тачку се види да генеришући програмски код на овакав начин, без неких додатних конфигурација се не може добити неки дужи програмски код који би зашао дубље у неке елементе граматике.

4.6 Ограничења

До сада, главни циљ је био само изгенерисати неки синтаксно исправан *MicroJava* програмски код без неке контроле како ће он изгледати. Другим речима, до сад се само ослањало на случајно генерисање програма надајући се најбољем. Како се *MicroJavaFuzzer* изводи из класе *ISLaSolver*, може као додатан параметар конструктора да прими и ниску карактера која представља ограничења дефинисана у *ISLa* (*Input Specification Language*) језику. Та ограничења омогућавају контролу над излазом фазера, односно шта тај излаз мора да испоштује како би фазер завршио са радом.

У корену пројекта у фолдеру *input*, поред спецификација лексер и парсер постоји и датотека која се зове *constraint*. Она у суштини служи за дефинисање ограничења и оно што је написано у њој се директно прослеђује конструктору класе *MicroJavaFuzzer*.

Једна функционалност је коришћење функције *str.len* која као аргумент прима нетерминал на који треба ограничење да се примени тако да ће тај нетерминал у излазној секвенци имати ту дужину. На пример покретање фазера са ограничењем *str.len(<_ident>)=14* даје излаз:

```
program x7y6Nn0kVmQO4l {}
```

ISLa омогућава и произвољно везивање услова помоћу логичких оператора и и или, тако да се може на прошло ограничење надовезати да сваки идентификатор, поред што је дужине 14, почиње словом A: *str.len(<_ident>) = 14 and str.at(<_ident>, 0) = "A"*. Овакво ограничење даје излаз:

```
program A3kXH30AlPA_Z6 {
}
```

У функцији *str.at*, други аргумент је позиција где одговарајући карактер треба да се нађе и индексирање почиње од 0. Постоји и функција *str.to.int()* која као аргумент прима

нетерминал и онда га претвара у цео број што може бити корисно када је потребно генерисати на пример бројеве веће од 1000: *str.to.int(<_number>) > 1000*. Ова функција се може искористити и на пример да бројеви буду дељиви са 3: *str.to.int(<_number>) mod 3 = 0*.
Осим *mod* оператора, постоје и стандардни аритметички оператори који се могу користити.

Овде сада настаје проблем. Док се користио нетерминал *<_ident>* за постављање услова, ти услови су били испуњени јер се у излазном програму морао наћи идентификатор као име програма. Међутим, ако се уведе ограничење као што су наведени примери за *<_number>* и не уведе неки услов да се сигурно изгенерише неки број, ограничење се неће ни видети. Једно решење за ово је коришћење функције *count(VARIABLE, NONTERMINAL, N)*. Она као аргументе прима променљиву за коју мора да важи да је број нетерминала у њој једнак *N*. Тако да се може увести ограничење:

```
count(start, "<_integer>", "1") and str.to.int(<_integer>) + 2 >
102,
```

које ће изгенерисати код:

```
program eC const SQk45dM_W1lrmjLaXxgG9cAVDy6Sz0sEG9kf ::
FfJB2tUqvhNie8oTOYFPuHbpI7RnwZK3s3 dXfO = 109 ;
{
}
```

Променљива *start* означава почетни нетерминал *<start>* који се имплицитно дефинише, а *NONTERMINAL* и *N* треба писати између наводника.

Постоји могућност да се контролише излаз тако што се може приступати неком елементу смене неког нетерминалног симбола. Ово је посебно значајно ако није пожељно применити ограничење на цео стартни симбол, већ само на развијање тог једног нетерминала. Ово се постиже уланчавањем тих нетерминалних симбола тачкама. На пример, циљ је постићи да име програма буде *prog*, и да постоји једна класа чије је име *klasa*, а да сви остали идентификатори не буду ограничени. То се може постићи ограничењем:

```
<ProgramName>.<IDENT> = "IDENT(prog)" and
count(start, "<ClassDecl>", "1") and
<ClassDecl>.<ClassDeclStart>.<IDENT> = "IDENT(klasa)"
```

Које генерише код налик овом:

```
program prog class klasa {
     static O aZcE1ghPQd4FfoOKNBaVTWwst285v9lRry [] ,
AISA3CHDpneUGxJk7X0qYm6z_uMibjL9s ;
     wL3 :: HEhE Ea [] ;
     {
          void o () {
          }
     }
}
{
     void v () {
```

```
     }
     void y () {
     }
}
```

Такође, још једна могућност приступа поделементима неког нетерминалног симбола је преко нотације са две тачке '..' где ће у целом подстаблу нетерминалног симбола елементи наведени после две тачке испуњавати услов, а не само директна деца. Ограничење које изазива да сви идентификатори у оквиру декларација променљивих буду низ карактера *"main"* је:

```
<VarDecl>..<_ident> = "main" and
count(start, "<VarDecl>", "2")
```

И генерисан програмски код:

```
program Q class
A0ck3stNfAVbFwlpCDGqai1ZBy_Rme96QJ7LgTMzEd5j8uhxoIPrHvYSX4KW2OnUvx
th4 {
     static main main ;
     main main ;
}
{
}
```

Види се да идентификатори који представљају име програма и име класе су случајно генерисани, док у класи постоје две декларације променљивих (од којих је једна статичка) и обе имају и тип и име променљиве *"main"*.

Некад је потребно осигурати да се на барем један елемент примени неко ограничење. То се постиже *exists* квантификатором. Општи облик овог квантификатора гласи

```
exists TYPE VARIABLE in CONTEXT:
    (CONSTRAINT)
```

Где *TYPE* представља неки нетерминал, *CONTEXT* је контекст у ком треба да се примени ограничење и исто представља нетерминал, *VARIABLE* представља алијас за *TYPE* и може се изоставити, и *CONSTRAINT* је ограничење које ће се применити само у дефинисаном контексту у формату који генерално важи за ограничења. Пример који ће илустровати ово јесте генерисање методе (аргумент командне линије је *start_symbol=<MethodDecl>*) где је барем један идентификатор дужине веће од 10. Поставља се ограничење:

```
exists <_ident> id in <start>:
    str.len(id) > 10
```

И на овај начин је генерисан код који :

```
c JQ3wIuY1WwfV (y :: AfBYnx uVTq ) {
     return (- false ) / n () - 'c' ;
}
```

Ако се изостави контекст, узима се да је <*start*> подразумевани контекст. Осим овог квантификатора, постоји и квантификатор *forall* који обезбеђује да сви елементи у контексту испуњавају наведено ограничење и има исти формат као и *exists* квантификатор.

Опет ће се генерисати само метода, али сада са ограничењем да сви идентификатори унутар формалних параметара методе буду дужи од 10. У ограничење је остављена и контрола дужине нетерминала <*FormPars*>, како фазер не би стално бирао празну секвенцу да брже конвергира ка решењу. Ограничење је:

```
str.len(<FormPars>) > 35 and
forall <_ident> id in <FormPars>:
     str.len(id) > 10
```

И овим ограничењем је добијен програмски код где се лако уверава да су оба идентификатора која се појављују у дефиницији формалних параметара стварно дужине веће од 10:

```
void R (dGcUs4wypKX Im37f306wnjiS ) fZ8F9tf :: HmEg v1rl , VQ2hV []
;
{
     for (;
     false ;
     ) continue;
}
```

4.7 Смернице

Приликом приче о ограничењима, неки од примера нису генерисали целе *MicroJava* програме, већ само неки део (углавном дефиницију метода). Како је сама граматика за апстрактно синтаксно стабло јако обимна, а фазер ради по принципу случајног избора коју ће смену да развије, када се уведу нека ограничења, он ће покушавати да развија смене изнова и изнова све док то неко ограничење не испуни.

Постоји у класи *ISLaSolver* уграђен систем који детектује када је неко ограничење немогуће испунити, међутим, наравно оно није идеално. Један једноставан пример ће ово приказати. Нека се уведе ограничење такво да у генерисаном *MicroJava* програмском коду постоји само један идентификатор и само једна декларација методе. Ограничење за то би изгледало:

```
count(start, "<MethodDecl>", "1") and count(start, "<IDENT>", "1")
```

При покретању скрипте са подразумеваним почетним симболом, *ISLaSolver* не детектује да је ово ограничење немогуће испунити па покушава да пронађе излаз који одговара улазној граматици и наведеном ограничењу. На овом примеру је очигледно видети зашто је немогуће испунити ово ограничење, а то је зато што *MicroJava* програмски код мора да има барем један идентификатор који је у ствари име програма, а када се уведе ограничење да мора да постоји и једна метода, аутоматски се каже да мора да постоји још минимално један

идентификатор који ће представљати име методе, што се директно коси са форсирањем да у целом програмском коду постоји само један идентификатор.

Приликом израде је одлучено да се само хватају и обрађују интерни изузеци који се бацају услед на пример изградње стабла или писања класа синтаксних чворова у датотеке. Остали изузеци који могу да се баце се директно прослеђују као излаз саме скрипте како би корисник имао увид шта се дешава. Под те остале изузетке углавном се мисли на изузетке које сам *ISLaSolver* баца, јер је боље да их корисник види у оригиналном облику.

Када класа *ISLaSolver* баци изузезак *StopIteration*, то је сигнал кориснику да се ниједно решење не може пронаћи, што значи да је детектовано да је дефинисано ограничење немогуће испунити.

Један од честих изузетака који се јављају јесте приликом коришћења функције *count* у дефинисању ограничења. Тамо је напоменуто да и нетерминал који се броји и број *N* се пишу између наводника. Ако се број не напише између наводника, баца се *AssertionError*:

```
    assert isinstance(num, DerivationTree)
AssertionError
```

Док са друге стране, ако се нетерминал не напише између наводника, тада се посматра као да цео тај део ограничења не постоји. Ако се пак напише нетерминал који не постоји у граматици онда се такође баца *AssertionError*:

```
    assert to_node in graph.all_nodes
AssertionError
```

У случају да се приликом покретања скрипте дефинише стартни симбол као неки који не постоји у граматици: *python main.py start_symbol=<abc>*, баца се *AssertionError*:

```
    assert start_symbol in grammar
AssertionError
```

Сви остали изузеци на које се може наићи се углавном јављају ређе, а и из приложених се може закључити да у самом изузетку пише довољно информација да корисник закључи шта се десило.

Постоји још један изузетак коме треба посветити пажњу:

```
TypeError:
evaluate_z3_expression.<locals>.not_implemented_failure() takes
from 0 to 1 positional arguments but 2 were given
```

Он се јавља када се искористи нека функција која постоји дефинисана у *SMT-LIB (Satisfiability Modulo Theories Library)* [6], а није имплементирана у класи *ISLaSolver*. Пример овакве функције јесте функција `contains`:

```
str.contains(<_ident>, "abc") = true
```

где се овде покушава рећи да сваки идентификатор мора да садржи подниску карактера *abc*, што би било јако корисно форсирати у ограничењу, али је ипак *ISLaSolver* са те стране лимитиран.

Већ је поменуто да се у неким примерима приликом примењивања ограничења прешло да стартни симбол буде <*MethodDecl*> како би се олакшало фазеру да примени ограничења. Када су та ограничења покренута над подразумеваним стартним симболом, требало је доста времена да се изгенерише излаз што је у већини случајева недопустиво за примену. Оволико чекање може настати из разлога што се тражи да доста услова буде испуњено у исто време, па је једна препорука да се не форсира много услова заједно у исто време (наравно овде велику улогу игра и граматика која у случају овог рада има поприлично велики број смена).

Приликом коришћења функције *count* треба обратити пажњу на број појављивања неке смене, јер ако се стави неки велики број, може довести до предугог чекања за генерисање излаза. Приликом израде самог фазера за програмски језик *MicroJava* и покушавања различитих ограничења како би се дошло до неких закључака како ће се фазер понашати у различитим ситуацијама, примећено је да контролисање броја <*Statement*> елемената доводи до мало дужих излаза који испитују доста елемената граматике (посебно јер сам нетерминал <*Statement*> има доста смена). Међутим, на машини на којој је тестирано је тај број био максимално 7 и то је доводило до довољно брзог генерисања излаза. Све преко тога изазива предуго чекање које није исплативо.

Добра идеја за генерисање целокупног *MicroJava* програма јесте да се сама <*Program*> смена:

```
Program ::= (Program) ProgramName NamespaceList GeneralDeclList
LEFT_BRACE MethodDeclList RIGHT_BRACE;
```

посматра из делова и да се сваки део засебно анализира и генерише посебно. Оно што је лоше код овог приступа јесте да сада постоји неки ручни напор који се мора уложити да би се генерисао неки пристојан *MicroJava* програмски код који је опет подоста мањи од ручног писања целог кода, а позитивна страна је што се могу увести строжа ограничења над самим елементима и тако заћи дубље у неке елементе граматике без размишљања о потрошеном времену на рад фазера.

5. Закључак

У овом раду је прилагођен већ постојећи фазер имплементиран као класа *ISLaSolver* како би се омогућило тестирање синтаксне анализе у *MicroJava* компајлеру. Приказано је како фазинг, као техника генерисања случајних улаза, може помоћи студентима да покрију различите сценарије и углове тестирања које не би лако приметили или превидели.

Главни проблем прилагођавања већ постојећег фазера јесте осмишљавање како на најбољи представити сам *MicroJava* код, односно како написати граматику која ће се проследити самом фазеру. Усвојено решење је да се не пише директно граматика самог програмског језика, већ да се осмисли репрезентација апстрактног синтаксног стабла као секвенце карактера, осмисли граматика за такву репрезентацију, искористи фазер да се изгенерише низ карактера и на основу њега изгради *AST*, на основу којег се изгенерише и сам *MicroJava* код. На крају су објашњена и ограничења као појам да се контролише сам излаз фазера како се не би потпуно ослонило на чисто случајно генерисање програмског кода.

Како сама класа *ISLaSolver* има могућност мутирања улаза, у смислу да ако неки улаз испуњава граматику којом је инстанца ове класе иницијализована, она може мутирати тај улаз што би значило да у њега уведе неке фази промене и тако генерише неки нови излаз који би доста личио на нешто што би човек направио. Ово је јако корисно код програмских кодова, из разлога што је јако тешко направити да излаз фазера буде било шта семантички исправно, па би мутације довеле до јако приближних семантички исправних програма. Тако да би ово била једна потенцијална тачка проширења овог рада, где би се сада морао имплементирати и сам део компајлера који би зависио од *.lex* и *.cup* спецификација да би био конзистентан са самим *MicroJava* фазером. Његова улога би била да од улазног програмског кода направи апстрактно синтаксно стабло и његову репрезентацију која би била улаз за мутирање.

Битно је напоменути да, иако се овај рад фокусира искључиво на програмски језик *MicroJava*, овај фазер се ефективно може применити на произвољан програмски језик. Потребно је само прилагодити *.lex* и *.cup* спецификације тако да подржавају граматику неког другог језика.

Литература

[1] Бојић Драган, Вукасовић Маја, *Програмски преводиоци 1*, 2022.

[2] Ari Takanen, Jared D. Demott, Charles Miller, Atte Kettunen, *Fuzzing for Software Security Testing and Quality Assurance, Second Edition*, 2018.

[3] Bart Miller, *Project List* (https://pages.cs.wisc.edu/~bart/fuzz/CS736-Projects-f1988.pdf, 05.09.2024.)

[4] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler, *The Fuzzing Book*, (https://www.fuzzingbook.org, 09.09.2024.)

[5] Igor Tartalja, *Posetilac*, (https://rti.etf.bg.ac.rs/rti/ir4ps/predavanja/Projektni%20uzorci/23%20Posetilac.pdf, 08.09.2024.)

[6] *Satisfiability Modulo Theories Library* (https://smt-lib.org/index.shtml, 09.09.2024.)

[7] Dominic Steinhöfel, Andreas Zeller, *ISLa Input Specification Language* (https://rindphi.github.io/isla/, 08.09.2024.)

[8] Dominic Steinhöfel, Andreas Zeller, *Input Invariants* (https://publications.cispa.saarland/3596/7/Input%20Invariants.pdf, 08.09.2024.)

[9] Michael Sutton, Adam Greene, Pedram Amini, *Fuzzing: Brute Force Vulnerability Discovery*, 2007.

Списак скраћеница

AST	*Abstract Syntax Tree* (апстрактно синтаксно стабло)
MJVM	*MicroJava Virtual Machine* (*MicroJava* виртуелна машина)

Списак слика

Списак листинга

A. Спецификација лексера коришћена у раду

```
package rs.ac.bg.etf.pp1;

import java_cup.runtime.Symbol;


%%


%{
	// ukljucivanje informacije o poziciji tokena
	private Symbol new_symbol(int type) {
		return new Symbol(type, yyline+1, yycolumn);
	}

	// ukljucivanje informacije o poziciji tokena
	private Symbol new_symbol(int type, Object value) {
		return new Symbol(type, yyline+1, yycolumn, value);
	}

%}

%cup
%line
%column

%xstate COMMENT

%eofval{
	return new_symbol(sym.EOF);
%eofval}


%%

" "	{ }
"\b"	{ }
"\t"	{ }
"\r\n"	{ }
"\f"	{ }

"program"	{ return new_symbol(sym.PROGRAM, yytext()); }
"break"		{ return new_symbol(sym.BREAK, yytext()); }
"class"		{ return new_symbol(sym.CLASS, yytext()); }
"else"		{ return new_symbol(sym.ELSE, yytext()); }
"const"		{ return new_symbol(sym.CONST, yytext()); }
"if"		{ return new_symbol(sym.IF, yytext()); }
"new"		{ return new_symbol(sym.NEW, yytext()); }
"print"		{ return new_symbol(sym.PRINT, yytext()); }
"read"		{ return new_symbol(sym.READ, yytext()); }
"return"	{ return new_symbol(sym.RETURN, yytext()); }
"void"		{ return new_symbol(sym.VOID, yytext()); }
"extends"	{ return new_symbol(sym.EXTENDS, yytext()); }
"continue"	{ return new_symbol(sym.CONTINUE, yytext()); }
"for"		{ return new_symbol(sym.FOR, yytext()); }
"static"	{ return new_symbol(sym.STATIC, yytext()); }
"namespace"	{ return new_symbol(sym.NAMESPACE, yytext()); }
```

```
"+"          { return new_symbol(sym.PLUS, yytext()); }
"-"          { return new_symbol(sym.MINUS, yytext()); }
"*"          { return new_symbol(sym.ASTERISK, yytext()); }
"/"          { return new_symbol(sym.SLASH, yytext()); }
"%"          { return new_symbol(sym.PERCENT, yytext()); }
"=="         { return new_symbol(sym.LOGICAL_EQUALS, yytext()); }
"!="         { return new_symbol(sym.LOGICAL_NOT_EQUALS, yytext()); }
">="         { return new_symbol(sym.GREATER_OR_EQUALS, yytext()); }
">"          { return new_symbol(sym.GREATER, yytext()); }
"<="         { return new_symbol(sym.LESS_OR_EQUALS, yytext()); }
"<"          { return new_symbol(sym.LESS, yytext()); }
"&&"         { return new_symbol(sym.LOGICAL_AND, yytext()); }
"||"         { return new_symbol(sym.LOGICAL_OR, yytext()); }
"++"         { return new_symbol(sym.INCREMENT, yytext()); }
"--"         { return new_symbol(sym.DECREMENT, yytext()); }
";"          { return new_symbol(sym.SEMICOLON, yytext()); }
"::"         { return new_symbol(sym.DOUBLE_COLON, yytext()); }
","          { return new_symbol(sym.COMMA, yytext()); }
"."          { return new_symbol(sym.DOT, yytext()); }
"("          { return new_symbol(sym.LEFT_PAREN, yytext()); }
")"          { return new_symbol(sym.RIGHT_PAREN, yytext()); }
"["          { return new_symbol(sym.LEFT_BRACKET, yytext()); }
"]"          { return new_symbol(sym.RIGHT_BRACKET, yytext()); }
"{"          { return new_symbol(sym.LEFT_BRACE, yytext()); }
"}"          { return new_symbol(sym.RIGHT_BRACE, yytext()); }
"=>"         { return new_symbol(sym.FOREACH, yytext()); }
"="          { return new_symbol(sym.EQUALS, yytext()); }

"//"              { yybegin(COMMENT); }
<COMMENT> .       { yybegin(COMMENT); }
<COMMENT> "\r\n"  { yybegin(YYINITIAL); }

[0-9]+                              { return new_symbol(sym.NUMBER,
Integer.parseInt(yytext())); }
true|false                      { return new_symbol(sym.BOOL,
Boolean.parseBoolean(yytext())); }
\'.\'                           { return new_symbol(sym.CHAR, yytext().charAt(1));
}
[a-zA-Z][a-zA-Z0-9_]*   { return new_symbol(sym.IDENT, yytext()); }

. { System.err.println("Leksicka greska ("+yytext()+") u liniji
"+(yyline+1)+"("+(yycolumn+1)+")"); }
```

Б. Спецификација парсера коришћена у раду

```
package rs.ac.bg.etf.pp1;

import java_cup.runtime.*;
import org.apache.log4j.*;
import rs.ac.bg.etf.pp1.ast.*;

parser code {:

    Logger log = Logger.getLogger(getClass());
    boolean errorDetected = false;


    // slede redefinisani metodi za prijavu gresaka radi izmene teksta poruke

    public void report_fatal_error(String message, Object info) throws java.lang.Exception {
      done_parsing();
      report_error(message, info);
    }

    public void syntax_error(Symbol cur_token) {
        report_error("\nSintaksna greska", cur_token);
    }

    public void unrecovered_syntax_error(Symbol cur_token) throws java.lang.Exception {
        report_fatal_error("Fatalna greska, parsiranje se ne moze nastaviti", cur_token);
    }

    public void report_error(String message, Object info) {
        errorDetected = true;
        StringBuilder msg = new StringBuilder(message);
        if (info instanceof Symbol)
            msg.append (" na liniji ").append(((Symbol)info).left);
        log.error(msg.toString());
    }

:}


init with {:
    errorDetected = false;
:}

scan with {:
    Symbol s = this.getScanner().next_token();
    if (s != null && s.value != null)
        log.info(s.toString() + " " + s.value.toString());
    return s;
:}

terminal PROGRAM, BREAK, CLASS, ELSE, CONST, IF, NEW, PRINT, READ, RETURN, VOID, EXTENDS, CONTINUE, FOR;
terminal STATIC, NAMESPACE, PLUS, MINUS, ASTERISK, SLASH, PERCENT, LOGICAL_EQUALS, LOGICAL_NOT_EQUALS;
terminal GREATER_OR_EQUALS, GREATER, LESS_OR_EQUALS, LESS, LOGICAL_AND, LOGICAL_OR, INCREMENT, DECREMENT;
terminal SEMICOLON, DOUBLE_COLON, COMMA, DOT, LEFT_PAREN, RIGHT_PAREN, LEFT_BRACKET, RIGHT_BRACKET, LEFT_BRACE;
terminal RIGHT_BRACE, FOREACH, EQUALS;
terminal Integer NUMBER;
terminal Boolean BOOL;
terminal String IDENT;
terminal Character CHAR;

non terminal NamespaceList, GeneralDeclList, MethodDeclList, GeneralDecl;
non terminal ConstDecl, VarDecl, ConstAssignmentList;
non terminal ConstAssignment, VarDeclList, SingleVarDecl, StatementList, SingleTypeVarDeclList;
non terminal SingleTypeVarDecl, FormPars, FormParsList, SingleFormPar, DesignatorStatement;
non terminal Addop, Mulop, TermList, FactorList, Statement;
non terminal ActPars, ActParsList, ForStatement, DesignatorList, ManyDesignators;
```

```
non terminal DesignatorStatementList, SingleDesignator, Relop, CondTermList, ClassVarDeclList;
non terminal CondFactList, StaticVarDeclList, StaticInitializerList, ClassMethodsList;
non terminal StaticInitializer, DesignatorExtension, SingleDesignatorExtension, FinalForStatement;
non terminal StaticInitializerStart, IfStart, ElseStart, FirstForSemicolon, SecondForSemicolon;

non terminal rs.etf.pp1.symboltable.concepts.Obj Program, ProgramName, NamespaceName, Namespace, Designator;
non terminal rs.etf.pp1.symboltable.concepts.Obj ConstValue, MethodName, MethodDecl, FuncCallStart,
  DesignatorUnzipStart;
non terminal rs.etf.pp1.symboltable.concepts.Obj ClassDeclStart, ClassDecl, StaticVarDeclStart,
  OneStaticInitializerStart;
non terminal rs.etf.pp1.symboltable.concepts.Obj StaticVarDecl, DesignatorName;
non terminal rs.etf.pp1.symboltable.concepts.Struct Type, Factor, Term, Expr, OneActPar, StartExpr;
non terminal rs.etf.pp1.symboltable.concepts.Struct Condition, CondFact, CondTerm, ForLoopStart;

precedence left ELSE;


Program ::= (Program) ProgramName NamespaceList GeneralDeclList LEFT_BRACE MethodDeclList RIGHT_BRACE;


ProgramName ::= (ProgramName) PROGRAM IDENT:progName;


NamespaceList ::= (NamespaceDeclarations) NamespaceList Namespace
                | (NoNamespace) /* epsilon */;


Namespace ::= (Namespace) NamespaceName LEFT_BRACE GeneralDeclList LEFT_BRACE MethodDeclList RIGHT_BRACE
  RIGHT_BRACE;


NamespaceName ::= (NamespaceName) NAMESPACE IDENT:namespaceName;


GeneralDeclList ::= (GeneralDeclarations) GeneralDeclList GeneralDecl
                | (NoGeneralDecl) /* epsilon */;


GeneralDecl ::= (GenConstDecl) ConstDecl
            | (GenVarDecl) VarDecl
            | (GenClassDecl) ClassDecl;


MethodDeclList ::= (MethodDeclarations) MethodDeclList MethodDecl
                | (NoMethodDecl) /* epsilon */;


ConstValue ::= (ConstNumber) NUMBER
            | (ConstChar) CHAR
            | (ConstBool) BOOL;


ConstDecl ::= (ConstDecl) CONST Type:varType ConstAssignmentList SEMICOLON;


ConstAssignmentList ::= (ConstAssignments) ConstAssignmentList COMMA ConstAssignment
                    | (SingleConstAssignment) ConstAssignment;


ConstAssignment ::= (ConstAssignment) IDENT:varName EQUALS ConstValue:val;


Type ::=  (ScopeType) IDENT:scopeName DOUBLE_COLON IDENT:typeName
        | (NoScopeType) IDENT:typeName;


VarDecl ::= (VarDeclaration) Type:varType SingleTypeVarDeclList SEMICOLON
            | error SEMICOLON:l {: parser.report_error("Izvrsen oporavak do ; u liniji " + lleft, null); :};


SingleTypeVarDeclList ::= (SingleTypeVarDeclarations) SingleTypeVarDeclList COMMA SingleTypeVarDecl
                        | (OneTypeVarDecl) SingleTypeVarDecl
                        | error COMMA:l {: parser.report_error("Izvrsen oporavak do , u liniji " + lleft, null);
  :} SingleTypeVarDecl;


SingleTypeVarDecl ::= (ArrayDecl) IDENT:varName LEFT_BRACKET RIGHT_BRACKET
                    | (NonArrayDecl) IDENT:varName;


```

```
ClassDecl ::= (ClassDecl) ClassDeclStart LEFT_BRACE StaticVarDeclList StaticInitializerStart ClassVarDeclList
ClassMethodsList RIGHT_BRACE;


ClassVarDeclList ::= (ClassVarDeclList) VarDeclList;


ClassDeclStart ::= (ExtendedClassDeclStart) CLASS IDENT:className EXTENDS Type
                | error:l {: parser.report_error("Izvrsen oporavak do { u liniji " + lleft, null); :}
                |  (BaseClassDeclStart) CLASS IDENT:className;


MethodDecl ::= (MethodDecl) MethodName LEFT_PAREN FormPars RIGHT_PAREN VarDeclList LEFT_BRACE StatementList
RIGHT_BRACE;


MethodName ::= (TypeMethodName) Type:retType IDENT:methName
             |  (VoidMethodName) VOID IDENT:methName;


VarDeclList ::= (VarDeclarations) VarDeclList VarDecl
             | (NoVarDecl) /* epsilon */;


FormPars ::= (FormParams) FormParsList
             | error:l {: parser.report_error("Izvrsen oporavak do ) u liniji " + lleft, null); :}
             | (NoFormPars) /* epsilon */;


FormParsList ::= (FormParamsList) FormParsList COMMA SingleFormPar
                | (OneFormPar) SingleFormPar
                | error COMMA:l {: parser.report_error("Izvrsen oporavak do , u liniji " + lleft, null); :}
SingleFormPar;


SingleFormPar ::= (FormParNonArray) Type IDENT:varName
                | (FormParArray) Type IDENT:varName LEFT_BRACKET RIGHT_BRACKET;


DesignatorStatement ::= (DesignatorAssignment) Designator:dest EQUALS Expr:e
                      |   (DesignatorIncrement) Designator INCREMENT
                      |   (DesignatorDecrement) Designator DECREMENT
                      |   (DesignatorFuncCall) FuncCallStart ActPars RIGHT_PAREN
                      |   (DesignatorUnzip) DesignatorUnzipStart ManyDesignators ASTERISK Designator
                       RIGHT_BRACKET EQUALS Designator;


DesignatorUnzipStart ::= (DesignatorUnzipStart) LEFT_BRACKET;


StatementList ::= (StmtList) StatementList Statement
                | (NoStatement) /* epsilon */;


Statement ::= (DesignatorStmt) DesignatorStatement SEMICOLON
             | error SEMICOLON:l {: parser.report_error("Izvrsen oporavak do ; u liniji " + lleft, null); :}
             | (ReadStmt) READ LEFT_PAREN Designator RIGHT_PAREN SEMICOLON
             | (PrintSingleStmt) PRINT LEFT_PAREN Expr RIGHT_PAREN SEMICOLON
             | (PrintMultipleStmt) PRINT LEFT_PAREN Expr COMMA NUMBER RIGHT_PAREN SEMICOLON
             | (IfStmt) IfStart LEFT_PAREN Condition RIGHT_PAREN Statement
             | (IfElseStmt) IfStart LEFT_PAREN Condition RIGHT_PAREN Statement ElseStart Statement
             | (ErrorIfStmt) IfStart LEFT_PAREN error RIGHT_PAREN:l {: parser.report_error("Izvrsen oporavak do )
u linij'. " + 'left, null); :} Statement
             | (BreakStmt) BREAK SEMICOLON
             | (ContinueStmt) CONTINUE SEMICOLON
             | (ReturnVoid) RETURN SEMICOLON
             | (ReturnNonVoid) RETURN Expr SEMICOLON
             | (ForConditionStmt) ForLoopStart LEFT_PAREN ForStatement FirstForSemicolon CondFact
SecondForSemicolon FinalForStatement RIGHT_PAREN Statement
             | (ForNoConditionStmt) ForLoopStart LEFT_PAREN ForStatement FirstForSemicolon SecondForSemicolon
FinalForStatement RIGHT_PAREN Statement
             | (StmtBlock) LEFT_BRACE StatementList RIGHT_BRACE;


```

```
210 IfStart ::= (IfStart) IF;
211
212
213 ElseStart ::= (ElseStart) ELSE;
214
215
216 ForLoopStart ::= (ForLoopStart) FOR;
217
218
219 FirstForSemicolon ::= (FirstForSemicolon) SEMICOLON;
220
221
222 SecondForSemicolon ::= (SecondForSemicolon) SEMICOLON;
223
224
225 FinalForStatement ::= (FinalForStatement) ForStatement;
226
227
228 Expr ::= (NegativeExpr) MINUS StartExpr Term TermList
229         | (PositiveExpr) StartExpr Term TermList;
230
231
232 StartExpr ::= (StartExpr) /* epsilon */;
233
234
235 TermList ::= (TermsList) TermList Addop Term
236             | (NoTerm) /* epsilon */;
237
238
239 Term ::= (Term) Factor FactorList;
240
241
242 FactorList ::= (FactorsList) FactorList Mulop Factor
243             | (NoFactor) /* epsilon */;
244
245
246 Factor ::= (NumberFactor) NUMBER
247             | (CharFactor) CHAR
248             | (SubExprFactor) LEFT_PAREN Expr:e RIGHT_PAREN
249             | (BoolFactor) BOOL
250             | (DesignatorFactor) Designator
251             | (ArrayFactor) NEW Type LEFT_BRACKET Expr:e RIGHT_BRACKET
252             | (ClassFactor) NEW Type LEFT_PAREN ActPars RIGHT_PAREN
253             | (FuncFactor) FuncCallStart ActPars RIGHT_PAREN;
254
255
256 FuncCallStart ::= (FuncCallStart) Designator LEFT_PAREN;
257
258
259 Designator ::= (Designator) DesignatorName DesignatorExtension;
260
261
262 DesignatorName ::= (ScopeDesignatorName) IDENT:scopeName DOUBLE_COLON IDENT:name
263                 |  (NoScopeDesignatorName) IDENT:name;
264
265
266 DesignatorExtension ::= (DesignatorExtensions) DesignatorExtension SingleDesignatorExtension
267                     | (NoDesignatorExtension) /* epsilon */;
268
269
270 SingleDesignatorExtension ::= (SingleDesignatorExtensionClassField) DOT IDENT:fieldName
271                             | (SingleDesignatorExtensionArray) LEFT_BRACKET Expr RIGHT_BRACKET;
272
273
274 Addop ::= (Addition) PLUS
275         | (Subtraction) MINUS;
276
277
278 Mulop ::= (Multiplication) ASTERISK
279         | (Division) SLASH
280         | (Moduo) PERCENT;
281
282
```

```
283 ActPars ::= (ActParams) ActParsList
284         | (NoActPars) /* epsilon */;
285
286
287 ActParsList ::= (ActParamsList) ActParsList COMMA OneActPar
288                 | (OneActParam) OneActPar;
289
290
291 OneActPar ::= (OneActPar) Expr;
292
293
294 ManyDesignators ::= (ManyDes) DesignatorList COMMA
295                 | (NoManyDesignators) /* epsilon */;
296
297
298 DesignatorList ::= (DesignatorsList) DesignatorList COMMA SingleDesignator
299                     | (SingleDes) SingleDesignator;
300
301
302 SingleDesignator ::= (Des) Designator
303                     | (NoDesignator) /* epsilon */;
304
305
306 ForStatement ::= (ForStmt) DesignatorStatementList
307                 | (NoForStatement) /* epsilon */;
308
309
310 DesignatorStatementList ::= (DesignatorStmtList) DesignatorStatementList COMMA DesignatorStatement
311                         | (SingleDesignatorStmt) DesignatorStatement;
312
313
314 Condition ::= (Condition) CondTerm CondTermList;
315
316
317 CondTermList ::= (CondTerms) CondTermList LOGICAL_OR CondTerm
318                 | (NoCondTerms) /* epsilon */;
319
320
321 CondTerm ::= (CondTerm) CondFact CondFactList;
322
323
324 CondFactList ::= (CondFacts) CondFactList LOGICAL_AND CondFact
325                 | (NoCondFacts) /* epsilon */;
326
327
328 CondFact ::= (CondFactRelop) Expr:e1 Relop Expr:e2
329             | (CondFactNoRelop) Expr;
330
331
332 Relop ::= (LogEq) LOGICAL_EQUALS
333         | (LogNotEq) LOGICAL_NOT_EQUALS
334         | (Grt) GREATER
335         | (Gre) GREATER_OR_EQUALS
336         | (Lss) LESS
337         | (Lse) LESS_OR_EQUALS;
338
339
340
341 StaticVarDeclList ::= (StaticVarDeclarations) StaticVarDeclList StaticVarDecl
342                     | (NoStaticVarDeclaration) /* epsilon */;
343
344
345 StaticVarDeclStart ::= (StaticVarDeclStart) STATIC;
346
347
348 StaticVarDecl ::= (StaticVarDecl) StaticVarDeclStart VarDecl;
349
350
351 StaticInitializerStart ::= (StaticInitList) StaticInitializerList
352                         | (NoStaticInitList) /* epsilon */;
353
354
355 StaticInitializerList ::= (StaticInitializers) StaticInitializerList StaticInitializer
```

```
356                       | (NoStaticInitializer) StaticInitializer;
357
358
359 StaticInitializer ::= (StaticInitializer) OneStaticInitializerStart StatementList RIGHT_BRACE;
360
361
362 OneStaticInitializerStart ::= (OneStaticInitializerStart) STATIC LEFT_BRACE;
363
364
365 ClassMethodsList ::= (ClassMethods) LEFT_BRACE MethodDeclList RIGHT_BRACE
366                     | (NoClassMethods) /* epsilon */;
```