Mikrojava kompajler

Projekat iz predmeta “Programski prevodioci 1” na odseku za softversko inženjerstvo Elektrotehničkog fakulteta u Beogradu

Autor: Aleksa Vučković

Datum: 24.12.2023.

Postavka

Cilj projektnog zadatka je realizacija kompajlera za programski jezik Mikrojavu. Kompajler omogućava prevođenje sintaksno i semantički ispravnih Mikrojava programa u Mikrojava bajtkod koji se izvršava na virtuelnoj mašini za Mikrojavu. Sintaksno i semantički ispravni Mikrojava programi su definisani specifikacijom [MJ]. Programski prevodilac za Mikrojavu ima četiri osnovne funkcionalnosti: leksičku analizu, sintaksnu analizu, semantičku analizu i generisanje koda.

Leskička analiza je implementirana Jflex alatom sa odgovarajućom .lex specifikacijom.  
Sintaksna analiza je implementirana CUP alatom sa odgovarajućom .cup specifikacijom.  
Semantička analiza i generisanje koda implementirani su obilaskom apstraktnog sintaksnog stabla dobijenog parsiranjem.

Komande

Komande se izvršavaju iz korenog direktorijuma projekta (MJCompiler).

Za generisanje leksera na osnovu specifikacije koristi se sledeća komanda:

java -jar lib/JFlex.jar -d ./src/rs/ac/bg/etf/pp1 spec/mjlexer.lex

Za generisanje parsera na osnovu specifikacije koristi se sledeća komanda:

java -jar lib/cup\_v10k.jar -destdir src/rs/ac/bg/etf/pp1 -ast src.rs.ac.bg.etf.pp1.ast -parser MJParser -buildtree spec/mjparser.cup

Za kompajliranje mikrojava izvornog koda, potrebno je pokrenuti iz komandne linije klasu Compiler i proslediti kao prvi argument izvorni fajl, a kao drugi argument izlazni fajl.

java -classpath "./bin;./lib/\*" rs.ac.bg.etf.pp1.Compiler test/code/reader.mj test/program.obj

Generisani objektni fajl može se pokrenuti koristeći dostavljenu mikrojava virtuelnu mašinu sledećom komandom, u kojoj se test/program.obj može zameniti lokacijom objektnog fajla:

java -cp lib/mj-runtime-1.1.jar rs.etf.pp1.mj.runtime.Run test/program.obj

Test primeri

Test primeri su podeljeni u tri grupe:

1. Syntax -> Test primeri koji testiraju oporavak parsera od greške.
2. Semantics -> Test primeri koji testiraju detekciju semantičkih greški.
3. Code -> Test primeri koji predstavljaju ispravne, izvršive programe. Svaki test primer je imenovan po osnovnom konstruktu koji testira, i na kraju test primera u okviru višelinijskog komentara dat je očekivani ispis.

Svaka grupa ima svoj potfolder u okviru foldera test.

Novouvedene klase

MySymbolTableVisitor je klasa koja je modifikacija standardne implementacije SymbolTableVisitor, prilagođena načinu upotrebe tabele simbola u ovom rešenju.

**class MySymbolTableVisitor {…}**

MyTab je klasa koja dopunjuje i u nekim delovima menja ponašanje Tab klase.

**class MyTab extends Tab {**

Metoda za direktno ubacivanje Obj objekta u tabelu.

**static boolean insert(Obj obj);**

Metoda za ubacivanje Obj čvora u roditeljski opseg aktuelnog opsega.

**static boolean insertParentScope(Obj obj);**

Metoda koja vraća roditeljski opseg aktuelnog opsega.

**static Scope parentScope();**

Poziva Tab.init, a zatim dodaje Obj čvor za bool tip u Universe opseg, i prepravlja neke detalje u objektnim čvorovima predefinisanih metoda.

**static void init();**

Otvara novi opseg, i odmah uvezuje simbole u prosleđeni objekat, tako da opseg i objekat imaju istu referencu na SymbolDataStructure.

**void openScopeAndChain(Obj);**

**void openScopeAndChain(Struct);**

Radi isto što i Tab.dump(), ali je podrazumevan MySymbolTableVisitor.

**void dump(SymbolTableVisitor stv);**

**void dump();**

**}**

ConstructorIterator predstavlja iterator za konstruktore prosleđene klase.

**class ConstructorIterator {**

**ConstructorIterator(Struct cls); //creates an iterator for cls**

**void reset();**

**Obj next(); //returns the next constructor as an Obj of kind Meth, or null if end has been reached**

**}**

FormalParamIterator predstavlja iterator za formalne parametre prosleđene metode.

**class FormalParamIterator {**

**FormalParamIterator(Obj cls);**

**void reset();**

**Obj next();**

**}**

MethodIterator predstavlja iterator za virtuelne metode prosleđene klase.

**class MethodIterator {**

**MethodIterator(Obj cls);**

**void reset();**

**Obj next();**

**}**

Utils je klasa sa uslužnim metodama.

**class Utils {**

Vraća true ako metoda methDerived može da bude override metode methBase.

**boolean overrideCompatible(Obj methBase, Obj methDerived);**

Vraća true ako metode imaju identične tipove argumenata.

**boolean identicalArguments(Obj meth1, Obj meth2);**

Vraća true ako metoda meth može da se pozove sa stvarnim argumentima params.

**boolean invokable(Obj meth, List<Struct> params);**

Vraća tip prvog formalnog parametra metode meth.

**Struct firstParam(Obj meth);**

Vraća prvi konstruktor klase cls kom odgovara lista tipova argumenata (lista ne treba da sadrži this).

**Obj findConstructor(Struct cls, TypeList);**

Vraća true ako tip from može da se dodeli tipu to (proširuje osnovnu implementaciju u tome što  
 dozvoljava dodelu izvedene klase osnovnoj).

**Boolean assignableTo(Struct from, Struct to);**

Vraća true ako je sub podtip klase base.

**Boolean subclassOf(Struct sub, Struct base);**

Vraća plitku kopiju obj

**Obj clone(Obj obj);**

Vraća true ako je metoda konstruktor

**boolean isConstructor(Obj meth);**

Vraća veličinu jednog objekta klase cls (u bajtovima).

**Int getSize(Struct cls);**

Vraća broj lokalnih promeljivih metode, uključujući argumente.

**Int localVariableCount(Obj meth);**

Prepravlja skok na adresi patchAdr, tako da destinacija bude drugi prosleđeni argument.

**void codeFixup(int patchAdr, int destination);**

Generiše poziv adrese adr.

**void generateCall(int adr);**

Vraća String reprezentaciju obj i struct čvora.

**String objToString(Obj obj);**

**String structToString(Struct struct);**

**}**

Postoji i 3 klase koje su samo omotači oko standardnih javinih klasa (jer one izgleda ne mogu da se koriste kao tip atributa neterminala u CUP alatu).

**class MyInt** -> Omotač oko Integer

**class TypeList** -> Omotač oko List<Struct>

**class DesignatorList** -> Omotač oko List<Obj>

IMPLEMENTACIJA

Klasa za proveru semantičke ispravnosti (SemanticAnalyzer), kao i klasa za generisanje koda (CodeGenerator), sadrže dva polja koja definišu kontekst:  
 Obj currentClass -> null van definicije klase, a pokazuje na Obj objekat klase u toku njenog parsiranja.  
 Obj currentMethod -> null van definicije metoda, a pokazuje na Obj objekat metode u toku parsiranja.

currentClass == null => Globalni kontekst.  
 currentMethod == null => Globalni statički kontekst.  
 currentMethod != null => Globalni kontekst metode.

currentClass != null => Klasni kontekst.  
 currentMethod == null => Klasni kontekst polja.  
 currentMethod != null => Klasni kontekst metode.  
  
Pored toga, klasa za proveru semantičke ispravnosti sadrži sledeća polja:   
 Struct declLineType => Tip koji se deklariše u tekućoj liniji deklaracija.  
 int totalStatic => Ukupan broj statičkih promenljivih. (tj. Adresa prve sledeće slobodne adrese u static)  
 int fors => Broj trenutno ugnežđenih for petlji.  
 int constructorCount => Broj definisanih konstruktora u currentClass.  
 bool mainFound => Da li je pronadjena globalna main metoda bez argumenata.

Klasa za generisanje koda sadrži sledeća polja:  
 List<Integer> initializerAdresses => Adrese automatski generisanih metoda za inicijalizaciju static memorije  
 Stack<List<Integer>> breakFixups => Adrese na kojima treba prepraviti break skokove.  
 Stack<List<Integer>> continueFixups => Adrese na kojima treba prepraviti continue skokove.

Upotreba polja klase Obj za različite tipove objekata:

**Con**type -> tipadr -> vrednost  
level -> 0  
fpPos -> NO\_VALUE

**Var**type -> tip  
adr -> static adresa / stek pomeraj  
level -> 0 = statička, 1 = stek  
fpPos -> za level == 1 -> 0 = argument, 1 = lokalna promenljiva (redosled argumenata je poznat iz adr)  
 za level == 0 -> NO\_VALUE

**Type**type -> ostatak definicije  
adr -> vtable adresa (NO\_VALUE za primitivne tipove)  
level -> 0  
fpPos -> broj polja

**Meth**  
type -> povratni tip  
adr -> adresa metode  
level -> 0 = statička, 1 = virtuelna  
fpPos -> broj argumenata

**Fld**type -> tip  
adr -> pomeraj u odnosu na adresu objekta  
level -> 1  
fpPos -> NO\_VALUE

**Elem**type -> tip  
adr -> NO\_VALUE  
level -> NO\_VALUE  
fpPos -> NO\_VALUE  
//Elem se nikada ne nalazi u tabeli simbola, već se koristi samo za parsiranje izraza

**Prog**type-> struct čvor sa članovima globalnog opsega  
adr -> NO\_VALUE  
level -> 0  
fpPos -> NO\_VALUE

**Namespace**type -> struct čvor sa članovima opsega imena  
adr -> NO\_VALUE  
level -> 0  
fpPos -> NO\_VALUE

Ispod je dat opis akcija pri obilasku čvorova, za semantičku proveru i generisanje koda.

Generisanje koda za uslovne izraze je realizovano tako da svaki uslov ima rezultat boolean, u vidu vrednosti 0 ili 1 na esteku.

Prilikom deklaracije klase, metode, programa ili prostora imena, kada se otvara novi opseg, odmah se uvezuju simboli, odnosno referenca na strukturu simbola iz opsega se kopira u odgovarajući Obj ili Struct čvor. Ovime se između ostalog postiže i uniformnost u parsiranju pristupa poljima preko referenci (ako se polju klase pristupa unutar njene deklaracije, ali ne preko implicitnog this nego preko konkretne reference), i u pristupu članovima prostora imena (ako se unutar prostora imena taj prostor imena eksplicitno imenuje).

Namespace je predstavljen čvorom Obj.Namespace koji sadrži ime, gde je Namespace konstanta definisana u Utils klasi, i čvorom Struct.None koji sadrži strukturu lokalnih simbola. Razlog zašto se koristi Struct jeste što Obj ne dozvoljava direktan pristup strukturi kao heš mapi, već samo kao kolekciji vrednosti, zbog čega bi pretraga bila neefikasna (a implementacija Obj ne može da se promeni, prema postavci). Takođe, čvorovima Obj.Prog, Obj.Class i Obj.Namespace se onda može uniformno pristupati kao opsezima imena.

Konstruktori se definišu kao statičke metode članice klase, sa specijalnim imenom \_\_constructorN, gde je N redni broj konstruktora. Ako nije definisan nijedan konstruktor, automatski se definiše podrazumevani konstruktor bez parametara, koji polja inicijalizuje nulama. Međutim, ovaj konstruktor se generiše tek nakon što je analizirana cela klasa, što znači da u klasi koja nema definisan konstruktor nije moguće pozivati konstruktor unutar samih metoda klase.

Izvedena klasa ne nasleđuje konstruktore osnovne klase, i ne može da ih poziva.

**Program = ProgramName NamespaceList DeclList { MethodList }**

**ProgramName = PROGRAM IDENT**

Na ProgramName se kreira Obj.Prog čvor i Struct.None čvor, i dodaje u tabelu. Ako ime već postoji, prijavljuje se greška.  
Obj referenca se pamti kao atribut u ProgramName i Program.  
Otvara se novi scope i simboli se odmah uvezuju.

Na Program se zatvara scope.  
Ako main metoda nije pronađena (mainFound == false), prijavljuje se greška.

Na Program se u initializerPc upisuje pc, a zatim se generišu enter 0,0 i pozivi svih inicijalizacionih blokova iz list initializerAddresses. Na kraju se generiše poziv adrese main metode (na osnovu mainPc), i nakon toga exit, return.

**Namespace = NamespaceName { DeclList { MethodList } }**

**NamepaceName = NAMESPACE IDENT**

Na NamespaceName se kreira Obj.Namespace čvor i Struct.None čvor. Obj se dodaje u tabelu. Ako ime već postoji, prijavljuje se greška.  
Obj referenca se pamti kao atribut u NamespaceName i Namespace.  
Otvara se novi scope i simboli se odmah uvezuju u Struct.

Na Namespace se zatvara scope.

**Decl = ConstDeclLine**

**| VarDeclLine**

**| ClassDecl**

**ConstDeclLine = ConstDeclLineStart ConstDeclComma;**

**ConstDeclLineStart = CONST Type**

**ConstDecl = IDENT = NUM**

**| IDENT = CHAR**

**| IDENT = BOOL**

Na ConstDeclLineStart se pamti tip linije.

Na ConstDecl se formira novi Obj.Con čvor sa odgovarajućom vrednošću i tipom. Tip mora da odgovara deklarisanom tipu. Obj se dodaje u tabelu. Ako ime već postoji, prijavljuje se greška.

Na ConstDeclLine se briše tip.

**ClassDecl = ClassName { StaticMemberList VarDeclLineList Methods }**

**ClassName = CLASS IDENT**

**| CLASS IDENT EXTENDS TYPE**

**Methods = { MethodList }**

|

Na ClassName, kreira se Struct.Class čvor i Obj.Type čvor sa imenom ident. Obj se dodaje u tabelu simbola. Ako ime već postoji prijavljuje se greška.   
U currentClass se pamti Obj, otvara se novi Scope i ulančavaju se simboliu struct.  
constructorCount = 0.  
  
U slučaju extends, proverava se da li je type klasa.  
1. Ako nije, prijavljuje se greška, a parsiranje se nastavlja kao da nema extends.  
2. Ako jeste, dohvataju se njeni locals. U scope se dodaju svi objekti, osim statičkih i konstruktora (level == 0). Ažurira se currentClass.fpPos tako da bude jednak osnovnoj klasi.

Na ClassDecl proverava se da li je deklaraisan makar jedan konstruktor. Ako nije, dodaje se Obj konstruktora bez argumenata koji inicijalizuje sva polja nulama.  
Scope se zatvara. Računa se veličina vtable, obilaskom svih metoda. Za svaku metodu se dodaje dužina njenog imena plus 2, i na kraju plus 1. U class.adr se upisuje totalStatic, a totalStatic se inkrementira za izračunatu veličinu. currentClass = null.

Na ClassName sa extends, adrese metoda osnovne klase se prepisuju u odgovarajuće objektne čvorove izvedene klase (a ako su overridovane, ove adresa će biti pregažena kada naiđe deklaracija nova implementacije).

Na ClassDecl prvo se provarava da li postoji podrazumevani konstruktor. Ako postoji, generiše se kod za njega kao sledeći niz instrukcija: enter 1, 1 – load 0 - const vtadr – putfield 0 – exit – return.  
Generiše se kod za popunjavanje vtable. U initializerAddresses dodaje se tekući pc na početak (kako bi se vtabele inicijalizovale pre izvršavanja statičkih inicijlizacionih blokova), a zatim se iterira kroz metode klase i generiše kod za popunjavanje static memorije počev od adrese zapisane u adr polju Obj objekta klase. Na kraju se generiše return instrukcija.

**StaticMember = StaticVarDeclLine**

**| StaticVarDeclLine**

**StaticVarDeclLine = StaticVarDeclLineStart StaticVarDeclList;**

**StaticVarDeclLineStart = STATIC Type**

**StaticVarDecl = IDENT**

**| IDENT[]**

Na StaticVarDeclLineStart se pamti tip.

Na StaticVarDecl se formira Obj.Var čvor sa imenom ident i odgovarajućim tipom, Kind = Var, level = 0, adr = totalStatic++. Obj se dodaje u tabelu. Ako ime postoji prijavljuje se greška.

Na StaticVarDeclLine se briše tip.

**StaticInitializer = StaticInitializerStart { StatementList }**

**StaticInitializerStart = STATIC**

Na StaticInitializerStart se trenutni pc dodaje u initializerAddresses na kraj. Generiše se enter 0,0.

Na StaticInitializer generiše se exit, return.

**Method = MethodDeclaration VarDeclLineList { StatementList }**

**MethodDeclaration = MethodName ( FormalParamComma )**

**MethodName = Type IDENT**

**| VOID IDENT** //Tab.noTyp

**| IDENT** //Tab.noTyp

Za globalni kontekst:

Na MethodName se kreira Obj.Meth sa imenom ident i deklarisanim tipom. Ako nije deklarisan tip, prijavljuje se greška i uzima void. Obj se čuva kao atribut u MethodName i MethodDeclaration. level=0 (statička), fpPos=0 (broj lokalnih promenljivih).

Otvara se novi scope, ulančava i nastavlja sa parsiranjem argumenata. currentMethod = obj.

Na MethodDeclaration, Obj se dodaje u tabelu, ali u parentScope. Ako ime postoji, prijavljuje se greška.

Na Method se scope zatvara. currentMethod = null.

Na MethodName se u Obj.adr upisuje trenutni pc. Generiše se instrukcija enter.

Na MethodDeclaration, ako je ime “main”, globalna metoda i bez argumenata, adresa se pamti u mainPc.

Na Method se generiše exit, return instrukcija ako je tip noTyp, a zatim i trap 1.

Za klasni kontekst:

Na MethodName se kreira Obj.ident sa imenom ident i deklarisanim tipom. Ako tip nije deklarisan, radi se o konstruktoru, i ime koje se dodeljuje je \_\_constructor + constructorCount. level = 0 za konstruktore, level = 1 za metode, a fpPos = 1 za sve (podrazumevani this).

Otvara se novi scope, ulančava, u njega se dodaje argument this, čiji se tip određuje na osnovu klasnog konteksta, i nastavlja se sa parsiranjem argumenata. currentMethod = Obj.

Na MethodDeclaration, u slučaju konstruktora, provarava se da li u currentClass postoji konstruktor sa istim parametrima. U ovu svrhu se koristi ConstructorIterator, FormalParamIterator i Utils::identicalArguments.  
1. Ako postoji, prijavljuje se greška, ali se nastavlja sa parsiranjem.  
2. U suprotnom dodaje se novi konstruktor u roditeljski scope, pri čemu se na ime konkatenira redni broj konstruktora. constructorCount++.

U slučaju nekonstruktora, Obj se dodaje u tabelu.  
1. Ako ime postoji, a nije metoda, prijavljuje se greška.  
2. Ako ime ne postoji, nastavlja se sa parsiranjem.  
3. Ako ime postoji i metoda je, proverava se da li je metoda override komaptibilna sa postojećom metodom, koristeći Utils::overrideCompatible().  
3.1. Ako nije, prijavljuje se greška, ali se nastavlja parsiranje.  
3.2. Ako jeste u postojeći Obj se kopira povratni tip i fpPos novokreiranog Obj, simboli iz aktuelnig scope-a se preulančavaju, a atribut obj MethodName i MethodDeclaration se menja postojećim Obj objektom.

NAPOMENA: Nije moguće menjati tip postojećeg Obj čvora, a nije moguće ni obrisati objektni čvor iz strukture podataka, tako da nije moguće u override metodi promeniti povratni tip.

Na Method se zatvara se scope. currentMethod = null.

Na MethodName se u Obj.adr upisuje trenutni pc. Generiše se enter.  
Ako je u pitanju konstruktor, generiše se instrukcija koja postavlja nulto polje na adresu vtable, čije je vrednost poznata iz currentClass.adr.

Na Method se generiše exit, return instrukcija ako je tip noTyp, a zatim i trap 1.

**FormalParam = Type IDENT**

**| Type IDENT[]**

Kreira se Obj sa imenom ident i odgovarajućim tipom. adr = currentMethod.fpPos++. level = 1, fpPos = 0 (argument). Obj se ubacuje u tabelu simbola. Ako postoji prijavljuje se greška.

**VarDeclLine = VarDeclLineStart VarDeclComma;**

**VarDeclLineStart = Type**

**VarDecl = IDENT**

**| IDENT[]**

Na VarDeclLineStart se pamti tip.

Na VarDecl se formira Obj čvor sa imenom ident i odgovarajućim tipom.

Za globalni statički kontekst: Kind = Var, level = 0, adr = totalStatic++.  
Za klasni kontekst polja: Kind = Field, level = 1, adr = ++currentClass.fpPos.   
Za kontekst metode: Kind = Var, level = 1, adr = currentMethod.fpPos++, fpPos = 1  
  
Obj se dodaje u tabelu. Ako već postoji, prijavjuje se greška.

Na VarDeclLine se briše tip.

**MethodInvocation = Designator(ActPars)**

**ActPars = ActPars ActPar**

**|**

**ActPar = Expr**

Na ActPar, prenosi se struct.

Na ActPars, u listu List<Struct> dodaje se struct iz ActPar.

Na MethodInvocation, proverava se Designator.obj.  
1. Ako nije tipa METH, prijavljuje se greška.  
2. Ako je Obj.level == 1, u listu List<Struct> dodaje se tip relevantne klase kao prvi parametar.  
3. Proverava se da li tipovi stvarnih argumenata odgovaraju metodi Designator.obj. Ako ne odgovaraju prijavljuje se greška.  
4. U MethodInvocation.struct se upisuje povratni tip metode.

Na MethodInvocation, ako je Designator.obj == Tab.ordObj ili Tab.chrObj, ne generiše se kod. Ako je Designator.obj == Tab.lenObj, generiše se arraylength  
U suprotnom ako je Designator.obj.level == 0, generiše se call instrukcija sa obj.adr.  
U suprotnom najpre se ponovo generiše kod za računanje reference nad kojom se metoda poziva (ponovnim obilaskom Designator podstabla), zatim se učitava njegovo prvo polje, nakon čega se generiše invokevirtual, sa karakterima iz imena Designator.obj.name.

**Statement = IF (IfCondition) IfStatement**

**| IF (IfCondition) IfStatement ELSE Statement**

**IfCondition = Condition**

**IfStatement = Statement**

Na IfCondition generiše se uslovni skok, a u atributu se pamti pc.

Na IfStatement, proverava se da li je roditeljski čvor UnmatchedIf ili MatchedIf.  
 1. Ako je MatchedIf, generiše se bezuslovni skok i u IfStatement se pamti pc.  
 2. Ako je UnmatchedIf, nema akcija.

Na UnmatchedIf, skok sa adrese iz IfCondition se prepravlja na osnovnu trenutnog pc:  
 IfCondition kod  
 const1  
 jne ifend  
ifstatement: ----------- (IfCondition.adr)  
 IfStatement kod  
ifend:

Na MatchedIf, skok sa adrese iz IfCondition se prepravlja na osnovu adrese u IfStatement, a skok sa adrese iz IfStatement prepravlja se na osnovu pc:  
IfCondition kod  
 const1  
 jne else  
ifstatement: ----------- (IfCondition.adr)  
 IfStatement kod  
 jmp ifend  
else: ----------- (IfStatement.adr)  
 Statement kod  
ifend:

**Statement = | DesignatorStatement;**

**| BREAK;**

**Proverava se da li je fors > 0. Ako nije prijavljuje se greška (break van for petlje).**

**U breakFixups se dodaje trenutni pc, i generiše se jmp 0.**

**| CONTINUE;**

**Proverava se da li je fors > 0. Ako nije prijavljuje se greška.**

**U continueFixups se dodaje trenutni pc, generiše se jmp 0.**

**| RETURN;**

**Proverava se da li je currentMethod.type == Tab.noTyp. Ako nije prijavljuje se greška (return iskaz bez izraza za metodu koja nije void).**

**Generiše se exit, return.**

**| RETURN Expr;**

**Proverava se da li tip Expr odgovara (assignableTo) currentMethod.type. Ako nije prijavljuje se greška.**

**Generiše se exit, return.**

**| READ ( Designator );**

**Proverava se da li je Designator.obj FIELD, ELEM ili VAR. Ako nije prijavljuje se greška.  
 Proverava se da li je Designator.obj.type int ili bool ili char. Ako nije prijavljuje se greška.**

**1. Ako je Designator.obj tipa int ili bool, generiše se read, Code.store(Designator.obj).  
 2. Ako je Designator.obj tipa char, generiše se bread, Code.store(Designator.obj).**

**| PRINT(Expr);**

**Proverava se da li je expr int, char ili bool.**

**1. Ako je Expr.struct int ili bool, generiše se const4, print.  
 2. U suprotnom se generiše const1, bprint.**

**| PRINT(Expr, NUM);**

**Proverava se da li je expr int, char ili bool.**

**1. Ako je Expr.struct int ili bool, generiše se const NUM, print.  
 2. U suprotnom se generiše const NUM, bprint.**

**| { StatementList }**

**Statement = FOR (ForEntry ; ForCondition; ForExit) Statement**

**ForEntry = DesignatorStatementComma**

**ForCondition = ConditionFactor**

**|**

**ForExit = DesignatorStatementComma**

Na ForExit, for++. Na For, for--.

Na ForEntry, pamti se trenutni pc kao atribut (forcondition labela). U breakFixups i continueFixups dodaje se nova prazna lista na kraj.

Na ForCondition generiše se uslovni skok na kraj petlje i iza njega bezuslovni na Statement.  
Trenutni pc se pamti kao atribut (forexit labela). \*Na prazan ForCondition prvo se generiše const1.

Na ForExit generiše se bezuslovni skok na forcondition i pamti se pc (forstatement labela).

Na For se generiše bezuslovni skok na ForExit. Na osnovu trenutnog pc i adresa zapamćenih u čvorovima potomcima, vrše se prepravke skokova.  
Takođe, iz breakFixups i continueFixups se skida lista sa vrha i vrše se prepravke.

ForEntry kod  
forcondition: ----------- (ForEntry.adr)  
 ForCondition kod  
 const1  
 jne forend  
 jmp forstatement  
forexit: ----------- (ForCondition.adr)  
 ForExit kod  
 jmp forcondition  
forstatement: ----------- (ForExit.adr)  
 Statement kod  
 jmp forexit  
forend:

**DesignatorStatement = Designator Assignop Expr**

**Ako Designator.obj nije FIELD, ELEM, ili VAR prijavljuje se greška.  
Ako nije Expr.struct.assignableTo(Designator.obj.type) prijavljuje se greška.**

**Generiše se Code.store(Designator.obj).**

**| MethodInvocation**

**Ako struct nije noTyp, generiše se pop instrukcija.**

**| Designator++**

**Designator mora biti FIELD, ELEM ili VAR, i mora biti tipa int.**

**Za VAR, FIELD i ELEM se generiše redom /, dup i dup2.  
Zatim se generiše Code.load(Designator.obj).  
Zatim se generiše const1 i add.  
Na kraju Code.store(Designator.obj).**

**| Designator --**

**Designator mora biti FIELD, ELEM ili VAR, i mora biti tipa int.**

**Za VAR, FIELD i ELEM se generiše redom /, dup i dup2.  
Zatim se generiše Code.load(Designator.obj).  
Zatim se generiše const1 i sub.  
Na kraju Code.store(Designator.obj).**

**| [ DesignatorComma \* Designator] = Designator**

**DesignatorComma sadrži kao atribut DesignatorList, listu Obj objekata koje zovemo elemObj. Srednji Designator sadrži kao atribut leftArrObj. Desni Designator sadrži kao atribut rightArrObj.  
Mora da važi:  
1. kind == Var ili Field ili Elem za sve objekte.  
2. leftArrObj.type.kind == Array   
3. rightArrObj.type.kind == Array  
4. rightArrObj.type.elemType je dodeljiv svakom elemObj.type  
5. rightArr.Obj.type.elemType je dodeljiv leftArrObj.type.elemType**

**U generisanom kodu koriste se privremen promenljive iza alociranog steka (u ovom slučaju bezbedno jer nema poziva funkcija), i to:  
pstackTop -> referenca na niz sa desne strane  
pstackTop+1 -> trenutni indeks niza sa desne strane  
pstackTop+2 -> trenutni indeks niza sa leve strane**

**Pri generisanju koda, računa se broj elemenata DesignatorComma niza kao varCount, i u generisanom kodu taj broj se koristi kao konstanta.  
Na početku generisanja adrese svih Designator objekata su naslagane na steku s leva na desno.  
  
Code.load(rightArrObj)  
store pstackTop  
Code.load(leftArrObj)  
dup  
arraylength //sada su na esteku referenca na levi niz i njegova dužina  
load pstackTop  
arraylength   
dup //sada su na esteku leftArr, len(left), len(right), len(right)  
const varCount  
jge 5  
trap 0 //broj designatora sa leve strane je veci od duzine niza sa desne**

**const varCount  
sub //broj elemenata desnog niza koji se smesta u levi niz  
dup2 //na esteku su leftArr, len(left), excess, len(left), excess  
jlt 9  
store pstackTop+2 //len(leftArr) je dovoljna za sve elemente viška  
pop  
jmp 6  
pop  
store pstackTop+2 //len(leftArr) nije dovoljna za sve elemente viška  
  
load pstackTop+2  
const varCount  
add  
store pstackTop+1 //indeks desnog niza je indeks levog plus varCount  
  
load pstackTop+2  
const0  
jeq 21  
  
inc pstackTop+2, -1  
inc pstackTop+1, -1  
dup  
load pstackTop+2  
load pstackTop  
load pstackTop+1  
(b)aload  
(b)astore  
jmp -21**

**pop  
for obj in desingatorList, in reverse order:  
 dec pstackTop+1  
 if obj != null:  
 load pstackTop  
 load pstackTop+1  
 (b)aload  
 Code.store(obj)**

**Condition = Condition || ConditionTerm**

**Generiše se add.**

**Skok na adresi koja je zapamćena u Conditon, prepravlja se tako da skače na tekuću adresu.**

**Proverava se da li je roditeljski čvor ConditionTermAnd. Ako jeste, generiše se:  
dup  
const\_1  
jeq x**

**Adresa x se popunjava naknadno u roditeljskom čvoru, a adresa instrukcije jeq se pamti u atributu ovog čvora.**

**Condition = ConditionTerm**

**Proverava se da li je roditeljski čvor ConditionOr. Ako jeste, generiše se:  
dup  
const\_1  
jeq x**

**Adresa x se popunjava naknadno u roditeljskom čvoru, a adresa instrukcije jeq se pamti u atributu ovog čvora.**

**ConditionTerm = ConditionTerm && ConditionFactor**

**Generiše se mul.**

**Skok na adresi koja je zapamćena u ConditonTerm, prepravlja se tako da skače na tekuću adresu.**

**Proverava se da li je roditeljski čvor ConditionTermAnd. Ako jeste, generiše se:  
dup  
const\_0  
jeq x**

**Adresa x se popunjava naknadno u roditeljskom čvoru, a adresa instrukcije jeq se pamti u atributu ovog čvora.**

**ConditionTerm = ConditionFactor**

**Proverava se da li je roditeljski čvor ConditionTermAnd. Ako jeste, generiše se:  
dup  
const\_0  
jeq x**

**Adresa x se popunjava naknadno u roditeljskom čvoru, a adresa instrukcije jeq se pamti u atributu ovog čvora.**

**ConditionFactor = Expr Relop Expr**

**Proverava se da li su tipovi izraza kompatibilni (isti, odnosno jedan null a drugi referencijalni).  
Ako je Expr.struct.isRefType(), Relop mora da bude == ili !=.**

**Generiše se:  
 jcc + relop.integer 7  
 const\_0  
 jmp 4  
 const\_1**

**| Expr  
 Generiše se:  
 const\_0  
 jne 7  
 const\_0  
 jmp 4  
 const\_1**

**Expr = Expr Addop Term**

**Tipovi moraju da budu intTyp.**

**Generiše se Addop.integer.**

**| Term**

**| - Term**

**Term.struct mora biti intTyp. Prepisuje se struct.**

**Generiše se neg.**

**Term = Term Mulop Factor**

**Tipovi moraju da budu intTyp.**

**Generiše se Mulop.integer.**

**Term = Factor**

**Factor = NUM**

**U Factor.struct se upisuje intTyp.**

**Generiše se Code.loadConst(num).**

**| CHAR**

**U Factor.struct upisuje se charTyp.**

**Generiše se Code.loadConst(char).**

**| BOOL**

**U Factor.struct upisuje se boolTyp.**

**Generiše se Code.loadConst(bool).**

**| Designator**

**Proverava se da li je Designator FIELD, VAR, ELEM ili CON. U struct se upisuje Designator.obj.type.**

**Generiše se Code.load(Designator.obj)**

**| MethodInvocation**

**Ako je struct noTyp prijavljuje se greška (void metoda u izrazu). U suprotnom se struct upisuje.**

**Ne generiše se ništa.**

**| new Type [ Expr ]**

**Tip Expr mora biti int. U struct se upisuje new Struct(Array, Type.struct).**

**Ako je Type.struct == charType, generiše se newarray 0, u suprotnom newarray 1.**

**| (Expr)**

**| ConstructorInvocation (ActPars)**

**Na osnovu ConstructorInvocation.struct i ActPars.typelist pronalazi se odgovarajući konstruktor sa Utils::findConstructor. Ako ne postoji, prijavljuje grešku. U suprotnom u struct upisuje ConstructorInvocation.struct.**

**Pronalazi se konstruktor sa Utils::findConstructor. Poziva se call adr.**

**ConstructorInvocation = new Type**

**Type.struct mora da bude CLASS. U ConstructorInvocation.struct upisuje se Type.struct.**

**Generiše se new (Utils::getSize(Type.struct)) i dup.**

**Type = IDENT**

Pronalazi se iz tabele odgovarajući simbol. Ako ne postoji, ili nije simbol tipa, prijavljuje se greška, a koristi se noTyp.

**| IDENT::IDENT**

Pronalazi iz tabele prvo namespace objekat. Ako ne postoji prijavljuje grešku.  
Zatim među ćlanovima namespace objekta pronalazi traženo ime. Ako ne postoji, ili nije tip, prijavljuje grešku.

**Assignop = ASSIGN Assignop.integer = sym.ASSIGN**

**Relop = EQ Relop.integer = 0 (Code.eq)**

**| NE Relop.integer = 1**

**| GT Relop.integer = 4**

**| GE Relop.integer = 5**

**| LT Relop.integer = 2**

**| LE Relop.integer = 3**

**Addop = ADD Addop.integer = Code.add**

**| SUB Addop.integer = Code.sub**

**Mulop = MUL Addop.integer = Code.mul**

**| DIV Addop.integer = Code.div**

**| MOD Addop.integer = Code.mod**

**Designator = Designator DOT IDENT**

**| Designator [ Expr ]**

**| IDENT::IDENT**

**| IDENT**

**(3,4)  
1. U slučaju 3, prvo se pronalazi namespace, i ako ne postoji prijavljuje se greška.  
2. Pronalazi se ime u tabeli (4), odnosno namespace objektu (3). Ako ne postoji prijavljuje se greška, a u obj se upisuje noObj.  
3. Ako je pronađeni objekat nestatički (level == 1), a currentMethod == null, prijavljuje se greška. U suprotnom se u obj upisuje nađeni obj.**

**(2)   
1. Ako tip Expr nije int, prijavljuje se greška.  
2. Ako je Designator.obj.type nije Array, prijavljuje se greška.  
3. U obj se upisuje Obj(Elem, Designator.obj.type.elemType)**

**(1)  
1. Ako je Designator.obj tipa CON ili METH prijavljuje se greška.  
2. Ako je Designator.obj tipa TYPE, PROG ili NAMESPACE, onda se u njegovom Designator.obj.type.locals traži ident. Ako se ne nađe prijavljuje se greška. Ako se nađe, ali je level == 1, prijavljuje se greška. U suprotnom se nađeni objekat upisuje u obj.  
3. U suprotnom u Designator.obj.type.locals se traži ident. Ako se ne nađe prijavljuje se greška. Ako je pronađeni objekat tipa statički (level == 0) prijavljuje se greška. U suprotnom upisuje se u obj.**

**(4)   
1. Ako je obj (neterminala sa leve strane) level == 1, a nije Var, generiše se instrukcija koja učitava nulti argument (this) na estek.   
2. U ostalim slučajevima ne generiše se ništa.**

**(3)  
1. Ne generiše se ništa.**

**(2)  
1. Generiše se store koji smešta vrh esteka (indeks) na statičku adresu 0.  
2. Generiše se Code.load(Designator.obj).  
2. Na estek se učitava sačuvan indeks.**

**(1)  
1. Ako je Designator.obj Type, Prog ili Namespace ne generiše se ništa. Con i Meth nisu opcije.  
2. U suprotnom, generiše se Code.load(Designator.obj)**