**REPUBLIKA E SHQIPËRISË**

**UNIVERSITETI I TIRANËS**

**FAKULTETI I SHKENCAVE TË NATYRËS**

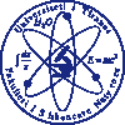
**DEGA: INFORMATIKË**

**Mikrotezë për mbrojtjen e gradës “Master i Shkencave” (MSC)**

**Tema: “Implementimi i njё Kompilatori pёr Algoritme tё Shprehura nё Pseudokod”**

**Punoi Udhëhoqi**

**Rando Shtishi Msc. Klesti Hoxha**

**REPUBLIKA E SHQIPËRISË**

**UNIVERSITETI I TIRANËS**

**FAKULTETI I SHKENCAVE TË NATYRËS**

**DEGA: INFORMATIKË**

**Mikrotezë për mbrojtjen e gradës “Master i Shkencave” (MSC)**

**Tema: “Implementimi i njё Kompilatori pёr Algoritme tё Shprehura nё Pseudokod”**

**Punoi Udhëhoqi Rando Shtishi Msc. Klesti Hoxha**

**(\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_) (\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_)**

**Tiranë, më \_\_\_\_/\_\_\_\_\_/\_\_\_\_\_**

PërmbajtJa

[1. Hyrje 5](#_Toc455267197)

[1.1. Përshkrimi i problemit 5](#_Toc455267198)

[1.2. Detyrat e kësaj teme 5](#_Toc455267199)

[2. Përshkrimi i literaturës 6](#_Toc455267200)

[2.1. Përshkrimi i literatures së grumbulluar për këtë teme 6](#_Toc455267201)

[2.1.1. Historia e evoluimit të gjuhëve të programimit 6](#_Toc455267202)

[2.1.2. Kompilatori 6](#_Toc455267203)

[2.1.3. Ndryshimet ndërmjet kompilatorëve dhe interpretuesve 7](#_Toc455267204)

[2.1.4. Etapat e kompilimit 8](#_Toc455267205)

[2.1.5. Fazat e kompilimit 8](#_Toc455267206)

[2.1.6. Mjetet për ndërtimin e kompilatorëve 9](#_Toc455267207)

[2.1.7. Analiza leksikore 9](#_Toc455267208)

[2.1.8. Shprehjet e rregullta 10](#_Toc455267209)

[2.1.9. Analiza Sintaksore 11](#_Toc455267210)

[2.1.10. Parserat 12](#_Toc455267211)

[2.1.11. Gramatikat 12](#_Toc455267212)

[2.1.12. Pemët e parsimit dhe derivimet 14](#_Toc455267213)

[2.1.13. Dykuptueshmëria ose Ambiguiteti 15](#_Toc455267214)

[2.1.14. Parserat e tipit top-down 16](#_Toc455267215)

[2.1.15. Parserat bottom-up 16](#_Toc455267216)

[2.1.16. Analiza semantikore 16](#_Toc455267217)

[2.1.17. Mjediset e ekzekutimit 18](#_Toc455267218)

[2.1.18. Pemet e aktivizimit 18](#_Toc455267219)

[2.1.19. Shpërndarja (alokimi) e kujtesës 19](#_Toc455267220)

[2.1.20. Tabela e simboleve 20](#_Toc455267221)

[2.1.21. Menaxhimi i shtrirjes së aksesit të ndryshorëve 21](#_Toc455267222)

[2.1.22. Kodi i ndërmjetëm 21](#_Toc455267223)

[2.1.23. Gjenerimi i kodit 22](#_Toc455267224)

[2.1.24. Optimizimi i kodit 22](#_Toc455267225)

[2.2. Dokumentacioni i teknologjive 23](#_Toc455267226)

[2.3. Punë të tjera në lidhje me këtë teme 29](#_Toc455267227)

[2.3.1. Kompilator për shprehjet aritmetike 29](#_Toc455267228)

[2.3.2. Kompilator për aksesimin e databazës së një videoteke 33](#_Toc455267229)

[3. Modelimi i zgjidhjes 40](#_Toc455267230)

[3.1. Konteksti i biznesit 40](#_Toc455267231)

[3.2. Rastet e përdorimit/kërkesat 40](#_Toc455267232)

[3.3. Arkitektura Logjike 41](#_Toc455267233)

[4. Zhvillimi 43](#_Toc455267234)

[4.1. Teknologjite e përdorura 43](#_Toc455267235)

[4.2. Detaje mbi zhvillimin 43](#_Toc455267236)

[4.3. Dokumenti i vendimeve kyçe të marra gjatë zhvillimit 82](#_Toc455267237)

[5. Rezultate 84](#_Toc455267238)

[5.1. Hyrje 84](#_Toc455267239)

[5.2. Instruksionet bazë 84](#_Toc455267240)

[5.3. Strukturat Përzgjedhëse 89](#_Toc455267241)

[5.4. Strukturat Përsëritëse 91](#_Toc455267242)

[6. Konkluzione 93](#_Toc455267243)

[7. Punë që mund të zhvillohen në të ardhmen 95](#_Toc455267244)

[8. Referenca ose Bibliografia 96](#_Toc455267245)

# 1. Hyrje

## Përshkrimi i problemit

Studentët e vitit të parë të degës informatikë dhe teknologji informacioni zhvillojnë një lendë që quhet hyrje në informatikë ku mësojnë të shkruajnë pseudokod për zgjidhjen e problemeve. Pseudokodi është gjuhë e thjeshtuar që përdoret për përshkrimin e konceptëve bazë të programit dhe të algoritmeve të ndryshëm. Pseudokodi nuk është bërë për t’u interpretuar nga makina . Pseudokodi është një konvencion që bën më të lehtë dizenjimin e algoritmeve dhe shpjegimin e logjikës së programit tek personat e tjerë .

Arsyet që e bëjnë pseudokodin më të kuptueshëm se gjuhët e mirëfillta të programimit janë:

* Bashkësia e instruksioneve të shkruara në gjuhën amtare e bejne më të lehtë për t’u mbajtur mend dhe për t’u kuptuar.
* Sintaksa e thjeshtë e pseudokodit, e zhveshur nga komplikimet sintaksore që shoqërojnë gjuhët e programimit .

Duke qënë se nuk ekziston një intepretues/kompilator për pseudokodin, ekzekutimi i një programi të shkruar në pseudokod nuk është i mundur.

## Detyrat e kësaj teme

Detyra e kësaj teme është ndërtimi i një kompilatori që do të ekzekutojë programin e shkruar në pseudokod . Një nga detyrat e kësaj teme është edhe të formalizojë sintaksën e pseudokodit. Duke qënë se pseudokodi është përdorur për të skicuar zgjidhjen dhe jo implementimin e saj , nuk ka sintaksë të qartë. Një nder punët kryesore të kësaj detyre do të jetë përcaktimi i qartë i sintaksës së pseudokodit.

Arsyet e përdorimit të pseudokodit si nga programuesit me eksperience dhe ata fillestarë është për shkak të lehtësisë së skicimit dhe përshkrimit të idesë së tyre për zgjidhjen e problemit apo implementimin e një algoritmi. Një detyrë e rëndësishme e kësaj temë është te ruajë thjeshtësinë që ofron pseudokodi në shkrimin e zgjidhjeve të problemit në programe.

# Përshkrimi i literaturës

## Përshkrimi i literatures së grumbulluar për këtë teme

### Historia e evoluimit të gjuhëve të programimit

Kompjuterat e parë që dolën në vitet 1940 ishin të programuar në gjuhën makinë, një sekuencë 0 dhe 1-shash që udhëzonin kompjuterin se cfarë operacionesh duhej të ekzekutonte dhe në cfarë radhe. Operacionet ishin në nivel shumë të ulët , si kalimi i të dhënave nga një pozicion në një tjetër pozicion në kujtesë, mbledhja e dy vlerave të regjistrave, krahasimi i dy vlerave dhe kështu me rradhë. Ky lloj programimi ishte shumë i ngadaltë , acarues dhe plot me gabime. Këto programe pas shkrimit të tyre ishin shumë të vështira për t’u kuptuar dhe për t’u modifikuar[1].

Hapi i parë në ndërtimin e gjuhëve të programimit më afër gjuhëve të nivelit të lartë ishte zhvillimi i gjuhës asembler në vitin 1950. Instruksionet e gjuhës asembler ishin një përfaqësim me simbole i instruksioneve të makinës[1].

Në mes të viteve 1950 u zhvilluan gjuhët e programimit si FORTRAN për llogaritjet shkencore, COBOL për procesimin e të dhënave të biznesit dhe lisp për llogaritjet simbolike. Filozofia e krijimit të këtyre gjuhëve të programimit ishte krijimi gjuhës së nivelit të lartë që do të lehtësonte punën e programuesve në shkrimin e programeve të biznesit, programeve simbolike dhe llogaritjet numerike[1].

Sot kemi mijëra gjuhë programimi të cilat klasifikohen në shumë mënyra. Një mënyre klasifikimi është sipas gjeneratës. Gjenerata e parë e gjuhëve të programimit përfshin gjuhët e makinës. Gjenerata e dytë përfshin gjuhët simbolike si asembler. Gjenerata e tretë përfshin gjuhët e programimit te nivelit të lartë si FORTRAN,COBOL Lisp,C,C++ dhe JAVA. Gjenerata e katërt e gjuhëve të programimit përfshin gjuhë të dizenjuara për aplikime specifike si NOMAD për gjenerimin e raporteve, SQL për askesimin e databazes, PostScript për formatimin e tekstit. Gjuhët e gjeneratës së 5 janë gjuhët e bazuar në logjike dhe kushtëzime si Prolog dhe OPS5[1].

Një lloj tjetër klasifikimi për gjuhët e programimit i ndan gjuhët e programimit në imperative dhe deklarative. Gjuhët imperative janë gjuhët në të cilat programi specifikon se si do të kryhet llogaritja, ndërsa në gjuhët deklarative programi specifikon se cfarë do të kryhet. Gjuhët si C,C++,C# dhe JAVA janë gjuhë imperative. Gjuhët ML,Haskell dhe Prolog janë gjuhë deklarative[1].

### 2.1.2. Kompilatori

Kompilatori është një program që konverton gjuhët e nivelit të lartë në gjuhën asembler. Në mënyrë të ngjashme ,asembler është një program që konverton programe të shkruar në asembler në gjuhë makinë[2].

Marrim një shembull për të kuptuar se si ekzekutohet një program i shkruar në gjuhën C në një makinë duke përdorur kompilatorin e gjuhës C. Programuesi shkruan një program në gjuhen C që është një gjuhë e nivelit të lartë. Kompilatori i gjuhës C e perkthen atë në gjuhë të nivelit të ulët, në gjuhën asembler. Pastaj programi asembler e përkthen programin asembler në kod makinë ose kod objekt si njihet ndryshe. Linkeri është mjeti që përdoret për të lidhur së bashku pjesë të programit për të formuar kod makinë të ekzekutueshëm. Loader është mjeti që përdoret për ngarkimin e kodit makinë të ekzekutueshem në memorie për t'u ekzekutuar [2].

Preproçesori është pjesë e kompilatorit , është një mjet që prodhon të dhëna për kompilatorët . Merret me proçesimin e makrove, përfshirjen e skedarëve dhe zgjerimin e gjuhës [2].

Kompilatori ose interpretuesi është përgjegjës për konvertimin e programit nga gjuhë e nivelit të lartë në gjuhë të nivelit të ulët / gjuhën assembler [2].

Asembler është programi që përkthen gjuhën e shkruar në asembler në gjuhë makine. Rezultati pas përkthimit nga asembleri është skedari objekt që përmban një kombinim instruksionesh makinë dhe të dhëna të nevojshme për vendosjen e këtyre instruksioneve në memorie [2].

Linker është një program i kompjuterit që lidh dhe bashkon disa skedarë me kod objekt në menyrë që të krijojë një program të ekzekutueshëm. Detyra kryesore e linker është të kërkojë për module dhe rutina të referencuara në program dhe percaktojë vendin e memories se ku do te ngarkohen këto module dhe rutina [2].

Loader është pjesë e sistemit të operimit dhe është përgjegjës për ekzekutimin e skedarëve të ekzekutueshëm; llogarit madhësinë e programit, instruksionet dhe të dhënat dhe krijon hapësirë memorie për të [2].

### 2.1.3. Ndryshimet ndërmjet kompilatorëve dhe interpretuesve

**Kompilatoret kundrejt interpretuesve**

|  |  |
| --- | --- |
| Kompilatori merr në hyrje të gjithë programin. | Interpretuesi merr në hyrje vetëm një instruksion të programit. |
| Kompilatori gjeneron kod të ndërmjetëm të quajtur kod objekt. | Interpretuesit nuk gjenerojnë kod të ndërmjetëm/objekt. |
| Kompilatori ekzekuton strukturat përzgjedhëse shumë më shpejt se interpretuesit. | Interpretuesit i ekzekuton strukturat përzgjedhëse më ngadalë se kompilatorët. |
| Programet e kompiluara kërkojnë më shumë memorje , pasi i gjithë programi duhet të vendoset në memorie për t'u ekzekutuar. | Interpretuesit nuk gjenerojnë kod të ndërmjetëm dhe si rezultat programet e interpretuara janë me eficente në përdorimin e memories. |
| Gabimet raportohen pasi është kontrolluar i gjithë programi për gabime sintakse ose të tipeve të tjerë. | Gabimi raportohet në momentin e parë që haset dhe nuk vazhdon kontrollojë më tutje, derisa gabimi të regullohet.. |
| Programet e kompiluara , kompilohen njëherë dhe më pas mund të ekzekutohen shumëherë. | Programet e interpretuara kanë nevojë të interpretohen rresht për rresht sa herë që ekzekutohen. |
| Një gjuhë që përdor kompilator është një gjuhë më e vështirë për t'i gjetur gabimin. | Gjetja e gabimit është lehtë pasi interpretuesi ndalon dhe nuk vazhdon më tutje pa rreglluar gabimin. |
| Gjuhët që përdorin kompilator janë C,C++,COBOL etj. | Gjuhët që përdorin interpretues janë PHP,Visual Basic,Python,Rubby,Perl,Lisp |

Sqarimeve i ndryshimeve ndërmjet kompilatorëve dhe interpretuesve [3].

### 2.1.4. Etapat e kompilimit

Fazat në të cilat kompilimi ndahet në dy etapa. Etapat janë një grupim i fazave nëpër të cilat kalon kompilimi[4].

Etapa e analizes lexon programin , kodin burim dhe e ndan në pjesë më të vogla që kanë kuptim për programin , e organizon në strukturën e përcaktuar nga gramtika e programit dhe përkthen në përfaqësim të kodit të ndërmjetëm[4].

Etapa e dyte është sinteza , që gjeneron kodin makinë me ndihmën e kodit të ndërmjetëm dhe tabelës së simboleve[4].

### 2.1.5. Fazat e kompilimit

Faza e parë kompilimit njihet me emrin e analiza leksikore ose skanimi. Analizuesi leksikor lexon sekuencën me karaktere nga programi burim dhe i grupon këto karaktere në sekuenca karakteresh kuptimplotë të quajtura leksema. Për çdo lekseme , analizuesi leksikor prodhon si output një token të formës (emër i token, attribute-vlere) [1].

Faza e dytë është analiza sintaksore ose parsimi. Parseri përdor sekuencën e tokenave të prodhuara nga analiza leksikore për të krijuar një përfaqësim në formën e një peme që njihet me emrin pemë sintakse ose peme parsimi. Në këtë fazë tokenat janë organizuar në përshtatje me gramatikën e kodit burim [1].

Faza e tretë është analiza semantike .Analizuesi semantikor përdor pemët e sintaksës të prodhuara nga analizuesi sintaksor (parseri) dhe informacionin në tabelën e simboleve për të konstrolluar nëse programi është shkruar semantikisht saktë (ka kuptim ) dhe është ne kosistente me përcaktimin e gjuhës. Pas kësaj faze gjenerohen pemët e etiketuara të sintakesës që është nje pemë sintakse që ka vlerat e atributeve për çdo nyje te saj [1].

Tre fazat e para hyjnë tek etapa e analizimit që njihet ndryshe edhe si fazat e front-end. Fazat qe vijne pas saj jane etapa e sintezes ndryshe si fazat back-end[1].

Gjeneruesi i kodit të ndërmjetëm përdor pemët e etiketuara të parsimit të prodhuara nga analizuesi semantikor për të gjeneruar kodin e ndermjetem. Është një përfaqësim i kodit ndërmjet gjuhes te nivelit te larte dhe kodit makine. Kodi i ndermjetem duhet të gjenerohet në mënyrë që ta bëjë më të lehtë përkthimin në kod makine [1].

Optimizuesi i kodit është faza që bën optimizimin e kodit të ndërmjetëm. Optimizimi mund të jetë heqja e rreshtave kod që nuk janë të nevojshëm dhe organizmi i instruksioneve në mënyrë të tillë që përshpejtojnë ekzekutimin e kodit pa përdorur shumë burime të kompjuterit si CPU, memorie[1].

Gjeneruesi i kodit bën përfaqësimin e kodit të ndërmjetëm të optimizuar dhe përkthen në kod makine. Sekuenca instruksionesh të kodit makinë kryejnë detyrën ashtu siç është përcaktuar në kodin e ndërmjetëm[1].

Tabela e simboleve është një strukturë të dhënash që mban informacione për çdo ndryshore si emri i ndryshores, vlera e ndryshores dhe tipin e saj. Kjo strukturë të dhënash duhet të dizenjohet në mënyrë të tillë që kompilatori të aksesojë shpejt ndryshoren nëpërmjet emrit të saj dhe gjithashtu të ruajë vlera[1].

### 2.1.6. Mjetet për ndërtimin e kompilatorëve

Personat që merren me ndërtimin e kompilatorëve ashtu si çdo programues tjetër mund të përfitojnë nga ambjente të zhvillimit software që përmbajnë mjete si editor të gjuhës, debuggers,version controller,teste të automatizuara dhe etj.

Disa mjete që përdoren për ndërtimin e kompilatorëve përfshijnë :

Gjeneruesit e parserave që krijojnë analizuesit e sintaksës duke specifikuar gramatikën e gjuhës së programimit [1].

Gjeneruesit e skanuesve prodhojnë analizuesit leksikore nga specifikimi i shprehjeve të regullta për tokenat e gjuhës [1].

Motoret e përkthimit të sintaksës të orientuar, që prodhojnë bashkësi procedurash për bredhjen e pemës së parsimit dhe gjenerimit të kodit të ndërmjetëm [1].

Gjeneruesit e gjeneruesve të kodit që prodhojnë gjeneruesit e kodit nga bashkesia e regullave për përkthimin e çdo veprimi të gjuhës të ndermjetme në gjuhë makine [1].

Motorrët e analizës së rrjedhës së të dhënave që lehtësojnë mbledhjen e informacioneve se si vlerat transmetohen nga një pjesë e programimit te pjesët e tjera. Analiza e rrjedhës së te dhënave ka rol të rëndësishëm në optimizimin te kodit [1].

### 2.1.7. Analiza leksikore

Detyra kryesore e analizës leksikore është të lexojë sekuenca me karaktere të kodit burim e t’i grupojë ato në leksema dhe për çdo leksemë të kodit burim të prodhojë si dalje një sekuencë tokenash. Sekuenca e tokenave dërgohet më pas te faza e analizimit të sintaksës [1].

Është zakonshme që analizuesi i leksikut të ndërveprojë me tabelën e simboleve kur zbulon një identifikues. Kur gjen një identifikues si leksemë , duhet që atë leksemë ta vendosë në tabelën e simboleve. Në disa raste duhet të lexojë nga tabela e simboleve për të përcaktuar tokenin e duhur që i duhet kaluar parserit [1].

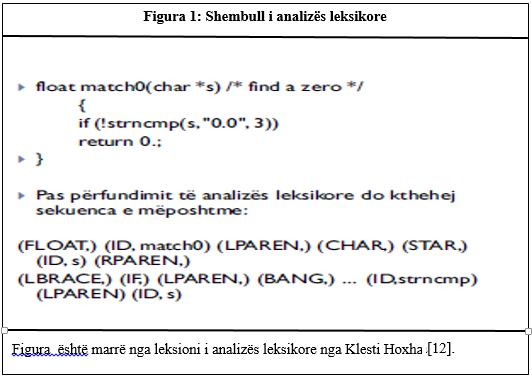
Analiza leksikore kryen dy procese:

* Procesi i skanimit si fshirja e komenteve, heqja e hapësirave [1].
* Procesi i analizës ku analizuesi prodhon një sekuencë tokenash si dalje të fazes [1].

Token është një çift i përbërë nga emri i tokenit dhe një çift attribute-vlere që është opsionale. Emri i tokenit është një simbol abstrakt që përfaqëson tipin e njësisë së leksemës. Emrat e tokenave janë simbolet që procesohen nga parseri [1].

Modeli është përshkrimi i formës që duhet të ketë një leksemë që të kategorizohet si një tip tokeni te caktuar. Tokenat e grupit fjalë kyçe kanë si model një sekuencë karakteresh që formojnë fjalën kyçe. Në rastin e identifikuesve dhe të disa tipeve të tjera tokenash, kanë një strukturë më komplekse që përputhen me disa sekuenca vargjesh me karaktere [1].

Leksema është një sekuencë karakteresh në kodin burim që përputhet me një model të një tokeni dhe që identifikohet nga analizuesi leksikor si instacë e asaj tokeni [1].



Kur kemi më shumë se një leksemë që përputhet me modelin e tokenit atëherë analizuesi leksikor duhet të japë informacion shtesë tek fazat pasuese të kompilimit për leksemen që përputhet me modelin. Për modelin e tokenit numrat 0 dhe 1 përputhen me modelin e tokenin numër , por është shumë rëndesishme për parserin që përveç emrit të tokenit duhet të dijë edhe vlerën e atributit që përshkruan leksemen e përfaqësuar nga tokeni [1].

### 2.1.8. Shprehjet e rregullta

Gjuha është e përbërë nga një bashkësi fjalësh dhe fjalët janë një sekuencë e fundme simbolesh. Dhe vetë simbolet që formojnë fjalët merren nga një alfabet i fundëm. Gjuha e fjalëve kyçe të C është bashkësia e të gjithë varg karaktereve në gjuhën C që nuk mund të përdoren si identifikues. Kur flasim për gjuhët në këtë mënyre ne përpiqemi të përcaktojmë n.q.s një fjalë e caktuar është apo jo pjesë e gjuhës. Për të përcaktuar këto gjuhë me një përshkrim formal përdorim shprehjet e rregullta[1].

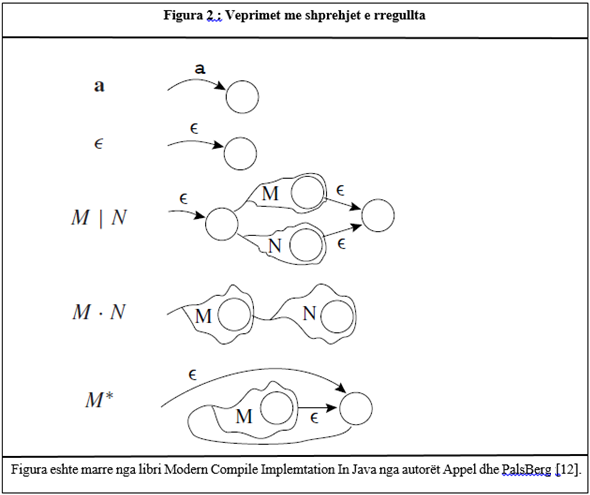
Simbol 🡪 për çdo simbol a të alfabetit, shprehja e rregullt përcakton një gjuhë që përmban vetëm fjalën a dhe epsilon[1].

Alternimi 🡪 ndërmjet dy shprehjeve të rregullta M dhe N , operatori i alternimit krijon një shprehje të rregullt të re M|N . Një varg karakteresh i përket gjuhës M | N n.q.s i përket gjuhë M ose i përket gjuhës N[1].

Konkatenimi 🡪 kur janë dy shprehje të rregullta M dhe N, operatori i konkatenimit krijon një shprehje të rregullt të re M N. Një varg karakteresh i përket gjuhës MN n.q.s përbëhet nga bashkëngjitja e dy vargjeve me karatere a dhe b , ku a bën pjesë te gjuha M dhe b bën pjesë te gjuha e N[1].

Epsilon 🡪 shprehja e rregullt që përfaqëson gjuhën ku element i vetëm është bashkësia boshe[1].

Përsëritja 🡪 për një shprehje të dhënë M , operatori i kleene është M\*. Një varg karakteresh që bën pjesë te gjuha M\* është bashkëngjitja e zero ose më shumë vargjeve me karaktere që janë pjesë e gjuhës M[1].



### 2.1.9. Analiza Sintaksore

Çdo gjuhë programimi ka rregulla të specifikuara që përshkruajnë strukturën sintaksore të programeve të shkruara saktë. Ndërtimi i sintaksës së një gjuhë programimi bëhet duke specifikuar gramatikat pa kontekst. Gramatikat pa kontekst përshkruajnë në mënyrë të qartë dhe të lehtë sintaksën e gjuhës së programimit. Gramatika lejon që gjuha të evolojë dhe të zhvillohet duke shtuar funksionalitete të reja për të kryer detyra të reja. Këto funksionalitete integrohen më lehtë në një implementim që ndjek rregullat e gramatikës që specifikojnë gramtatikën e gjuhës[1].

Analizuesi i sintaksës ose parseri merr tokenat e prodhuara nga analiza leksikore dhe verifikon nëse sekuenca e tokenave mund të gjenerohet nga gramatika e gjuhës së programimit. Për programet e shkruara saktë , parseri ndërton një pemë parsimi dhe kalon te faza tjetër e kompilimit për procesim[1].

### 2.1.10. Parserat

Kemi tre tipe parserash për gramatikat: universal,top-down dhe bottom-up [1].

Parserat universal janë ato parsera që janë në gjëndje të parsojnë çdo lloj gramatike pa kontekst. Por kjo vjen me një çmim , që janë shumë jo eficente për t’u përdorur në prodhimin e kompilatorëve [1].

Parserat top-down janë ato parsera që ndërtojnë pemën e parsimin duke filluar nga rrënja për tek gjethet [1].

Parserat bottom-up janë ato parsera që ndërtojmë pemën e parsimin duke filluar nga gjethet te rrënja [1].

Në secilin nga rastet informacioni që i vjen parserit skanohet nga majta në djathtë, me një simbol për kohë [1].

Për parserat top-down dhe bottom-up që janë eficente mund të parsojnë vetëm një klasë të gramatikave pa kontekst, specifikisht gramtikat LL dhe LR. Gramatika LL dhe LR mund të përshkruajnë sintaksën e një gjuhë programimi moderne. Zakonisht parserat që ndërtohen me dorë përdorin gramatikat LL, psh parseri parashikues përdoret mbi gramatikat LL. Parserat që përdoren mbi gramatikën LR janë një klase më madhe se klasa e gramatikes LL dhe ndërtohen me mjete të automatizuara [1].

### 2.1.11. Gramatikat

Një gramatikë pa kontekst (GPK) është një katërshe G = (N, A, P, S) ku

N është një bashkësi e fundme simbolesh jo fundore

A është një alfabet simbolesh fundore (i pavarur nga N)

P është një bashkësi e fundme rregullash prodhimi.

(production rules) të formës A -> a ku A ∈ N dhe a∈X ku X = N ∪ A

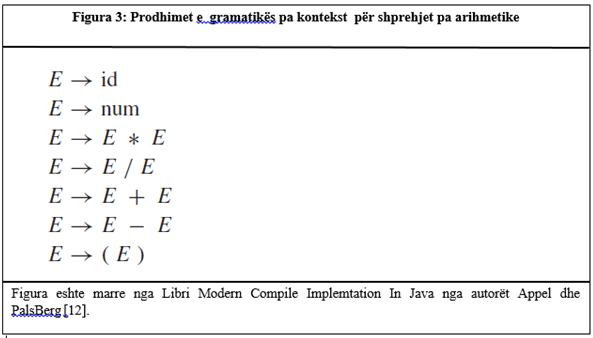
S është simboli fillestar (start simbol)

Fundorët janë simbolet bazë prej të cilave formojnë fjalitë. Termi emri tokenit është ekuivalent me me termin fundor [1].

Jo fundorët janë ndryshore sintakse që përcaktojnë një bashkësi me fjalë. Bashkësia e fjaleve të përcaktuara nga jo fundorët ndihmon në përcaktimin e gjuhës së gjeneruar nga gramatika. Jo fundorët imponojnë strukturën hierarkike të gjuhës e cila është çelësi i analizës sintaksore dhe përkthimit [1].

Në gramatikë , një jo fundor dallohet nga të tjerët si simboli fillestar dhe bashkësia me fjalë që përcakton është gjuha e gjeneruar nga gramatika [1].

Prodhimet ose produksionet specifikojnë mënyrën se si jo fundorët dhe fundorët kombinohen për të formuar fjali. Çdo produksion konsiston: në një jofundor që quhet koka, ose ana e majtë e produksionit, trupi ose ana e djathta e produksionit që konsiston në 0 ose më shumë fundorë dhe jo fundorë. Komponentët e trupit përshkruajnë mënyrën se si fjalitë e jo fundorëve mund të ndërtohen te koka [1].



Për të shmangur ngatërresën ndërmjet fundorëve dhe jo fundorëve kemi vendosur disa konveksione në simbolikën që përdoret për të përfaqësuar fundorët dhe jo fundorët [1].

Simbolika për fundoret përfshin:

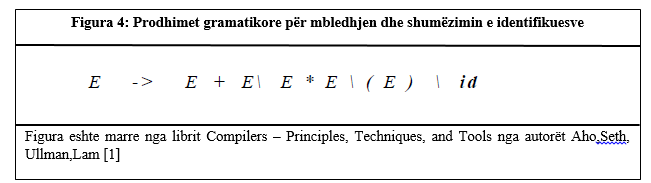
* Gërmat e vogla si a,b , c
* Operatorët +,\* etj
* Shenjat e pikësimit,kllapat dhe etj.
* Numrat 0-9
* Fjalët më të theksuara,

Simbolika për jo fundorët përfshin:

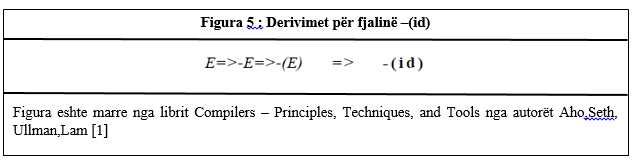
* Gërmat kapitale A,B,C
* Gërma S, që përdoret zakonisht për simbolin fillestar
* Fjalët me gërma të vogla të pjerrëta

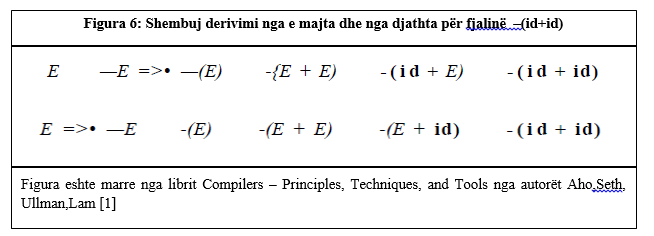
### 2.1.12. Pemët e parsimit dhe derivimet

Ndërtimi i pemëve të parsimit bëhet duke trajtuar prodhimet si rregulla rishkrimi. Fillimin me simbolin fillestar dhe me çdo hap të rishkrimit ,zëvëndësojme jo fundorin me trupin e prodhimit të fundorit [1].



Marrim si shembull gramatikën e mësipërme në figurën e katërt , me një simbol të vetëm jo fundor. Prodhimi E🡪 -E që E përcakton një shprehje. Zëvëndësimi i E me trupin e produksionit të saj bëhet duke shkruar –E në vend të E e cila lexohet . E derivon në –E. Prodhimi E 🡪 (E) mund të zbatohet me jofundor E që ndodhet në çdo fjalë të gramatikës së simboleve, E\*E, (E)\*E, ose E\*E🡪E\*(E). Ne mund të marrin një jo fundor të vetëm E dhe t’i zbatojmë prodhimet në mënyrë që të marrin një sekuencë zëvëndësimesh. Per shembull si në figurën e mëposhtme [1].

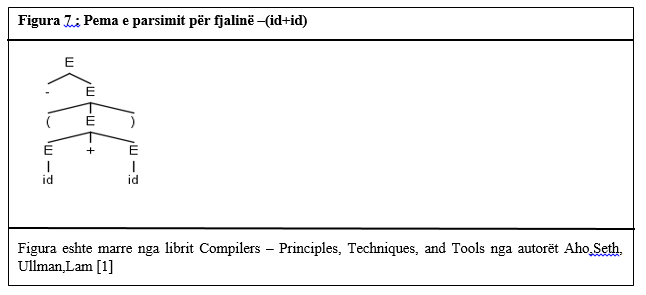


Derivimi është sekuenca e zëvëndësimeve që zbatohet për përfitimin e –(id) nga E. Ky derivim vërteton që fjalia –(id) është pjesë e instancës së shprehjes. Në çdo hap të derivimeve ne kemi dy mundësi zgjedhjeje . Duhet të zgjedhim se cilin jo fundor do të zëvëndësojmë me trupin e prodhimit të tij [1].

Çdo jo fundor zëvëndësohet me të njëjtin trup të prodhimit të tij, por ajo që ndryshon është radha e zëvëndësimeve. Për të kuptuar se si një parser funksionon duhet të marrim në konsideratë derivimet e gramatikës së gjuhës. Mënyrat se si jo fundorët zëvëndësohen jane:

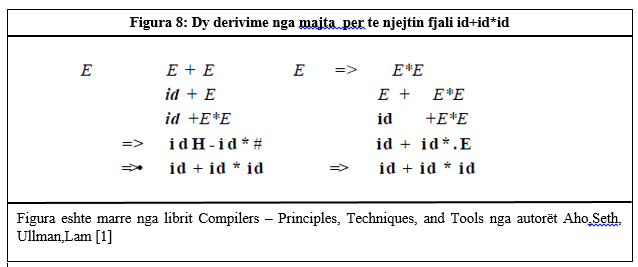
* Derivimet nga majta,janë derivimet ku gjithmonë simboli jo fundor në të majtë është ai që zëvëndësohet me trupin e prodhimit të tij.
* Derivimet nga e djathta janë derivimet ku gjithmonë simboli jo fundor në të djathtë është ai që zëvëndësohet me trupin e prodhimit të tij.

Një pemë parsimi është një përfaqësim grafik i derivimeve që filtron radhën në të cilën zbatohen prodhimet për zëvëndësimin e jo fundorëve. Çdo nyje e pemës së parsimit përfaqëson një zbatim të prodhimit. Nyja e brendshme është etiketuar me simbolin jofundor , kokën e prodhimit, ndërsa fëmija e kësaj nyje janë të etiketuara nga e majta në të djathë nga simbolet në trupin e prodhimit. Gjethet e pemës janë fundorët dhe lexohen nga majta në të djathtë për të formuar fjalinë që është pjesë e gjuhës të përcaktuar nga gramatika. Rrënja e pemës së parsimit është simboli fillestar [1].



### 2.1.13. Dykuptueshmëria ose Ambiguiteti

Kur një gramatikë që prodhon një ose më shumë pemë parsimi për një fjali të dhënë, atëherë themi që kjo gramatikë është ambuige. Një gramatikë ambuige është një gramatike që prodhon më shumë se një derivim nga e majta ose më shumë se një derivim nga e djathta për të njëjtën fjali [1].



Në figurën e mësipërme tregohet një gramatikë ambuige, pasi nga fjalia id+id\*id mund të gjenerohen më shumë se një derivim nga e majta. Kaq mjafton për të provuar që gramatika është ambuige.Për shumicën e parserave është e dëshirueshme që gramatika për parserin të mos jetë ambuige dhe kërkohet që të eliminohet ambiguiteti.Ndonjëherë një gramatikë ambuige mund të rishkruhet në një gramatikë që nuk është ambuige [1].

Për transformimin e gramatikave ambuige në gramatika jo ambuige përdorim teknikën e eliminimit të rekursionit në të majtë dhe eliminimit të faktorizimit në të majtë [1].

Një gramatikë ka rekursion në të majtë kur një jo fundor që ndodhet te koka e prodhimit ndodhet edhe ne fillim të trupit të prodhimit. Psh A🡪 Aa . Parserat top down nuk mund të përdoren me gramatika që kanë rekursion në të majtë. Prandaj duhen të bëhen transformime për eliminimin e rekursionit në të majtë [1].

Faktorizimi në të majtë është një transformin gramatikor që përdoret për prodhimin e gramatikave të përshtatshme për parserat top-down. Kur nuk kemi të qartë zgjedhjen që duhet bëjmë ndërmjet dy ose më shumë prodhimeve, atëherë duhet të rishkruajmë prodhimet në mënyrë që zgjedhja se me cilin prodhimi do të zëvëndësohet në varësi të leximit të inputit do të jetë e qartë [1].

### 2.1.14. Parserat e tipit top-down

Parserat top-down ndërtojnë pemën e parsimit nga fjalia , duke filluar nga rrënja dhe duke krijuar nyjet e pemës së parsimit në mënyrën e bredhjes pararendore. Në çdo hap të parsimit top-down , problemi kyç është përcaktimi i prodhimit që do të zbatohet mbi jo fundorët. Pasi është zgjedhur një prodhim për jo fundorin , i gjithë procesi i parsimit qëndron në përputhjen e fundorëve me trupin e prodhimit me inputin e tokenave hyrës. Klasa e gramatikave prej të cilave mund të ndërtojmë parserat top-down jane LL(K) [1].

### 2.1.15. Parserat bottom-up

Parserat bottom up fillojnë ndërtimin e pemës së parsimit nga gjethet derisa arrijnë te rrënja e pemës. Tek parserat bottom-up fillohet nga fjalia dhe aplikohen rregullat e prodhimit në mënyrë të kundërt për të arritur tek simboli fillestar.Parsimi bottom-up e redukton një fjali te simboli fillestar. Kjo arrihet duke përdorur reduktimet. Reduktimet janë e kundërta e prodhimeve. Reduktimet zëvëndësojnë krahun e djathtë të prodhimeve me jo fundorin në krahun e majtë [1]

### 2.1.16. Analiza semantikore

Pema e parsimit e ndërtuar gjatë fazës së analizës sintaksore nuk i nevojitet kompilatorit, sepse ajo nuk ka informacion se si të vlerësojë pemën e parsimit. Kur prodhimet e gramatikës pa kontekst nuk kanë rregulla semantikore të shoqëruara me to , nuk prodhojnë ndonjë rezultat të caktuar. Semantika e gjuhës përcakton kuptimin e gjuhës që ndërton. Semantika ndihmon në interpretimin e simboleve dhe tipeve të tyre dhe lidhjet me njëra tjetrën. Analiza semantikore kontrollon nëse struktura sintaksore e programit të ndërtuar ka kuptim [1].

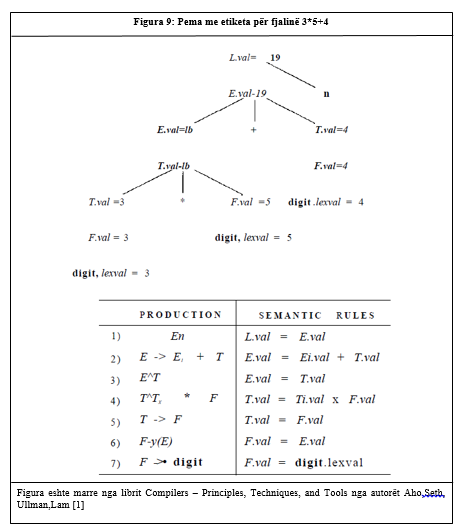
Gjatë kësaj faze të kompilimit ne shtojmë informacion mbi semantikën e programit në çdo nyje të pemës së parsimit. Zakonisht ky informacion që shtohet në nyjet e pemës së parsimit njihet si etiketa/specifika [1].

Marrim si shembull rregullat që përshkruajnë shprehjet aritmetike në gjuhët e nivelit të lartë. Për të trajtuar këto shprehje gjuhet e programimit duhet të specifikojnë :

* Tipin e rezultatit që do të kthejë shprehja(numer i plote, presje)
* Rezultatin e shprehjes

Në këtë mënyrë çdo simbol mund të ketë një tip dhe një vlerë të shoqëruar me të.

Analiza semantike merr si input pemën e parsimit të prodhuar gjatë fazës së analizës së sintakës dhe prodhon pemën e parsimit me etiketa ose specifika si më lart. Gramatika me atribute ose etiketa ose specifika është një lloj i vecantë gramatikash ku informacione shtesë i shtohen jo fundorëve për të dhënë informacion të ndjeshëm ndaj kontekstit. Siç thamë më sipër që pemës së parsimit i shtohen informacione shtesë te nyjet e pemës dhe informacioni që shtohet përcaktohet nga gramatika me specifika [1].



Në figurën e mësipërme tregohet gramatika me atribute , prej së cilës krijohet edhe pema e parsimit me atribute. Në djathtë të prodhimeve të gramatikës ndodhen rregullat semantike që specifikojnë se si gramatika duhet interpretuar [1] .

Përcaktimet sipas sintaksës (syntax-directed SDD) është një gramatikë pa kontekst me atribute dhe rregulla. Atributet shoqërojnë simbolet gramatikore, ndërsa rregullat shoqërojnë prodhimet [1].

Kemi dy tipe atributesh:

* Atributet e sintetizuara, vlera të cilave llogaritet nga vlera e nyjeve të bijve të saj.
* Atributet e trashëguara , vlera e të cilave llogaritet nga vlera e nyjeve prind të saj, ose të nyjeve të vellëzërve të tij.

Vlerësimi i një SDD (percaktimit sipas sintaksës) është ajo që na jep pemën e parsimit me atribute ose etiketa. Siç pamë më sipër pema e parsimit me atribute ose etiketa është një pemë që për çdo nyje të saj ka informacione shtesë [1].

Për të përcaktuar radhën e vlerësimit për instancat e atributeve në një pemë të dhënë përdoren grafet e varësisë. Grafi i varësisë përcakton se si do llogariten vlerat e atributeve të nyjes së pemës duke përcaktuar rrjedhën e informacionit midis instancave të atributeve të një pemë parsimi. Një brinjë e grafit nga një nyje e atributetit tek një tjetër do të thotë që vlera e së parës nevojitet për të llogaritur vlerën e nyjes së dytë[1].

Përcaktimet me atribute S janë përcaktime që kanë vetëm atribute të sintetizuara. Dhe për një përcaktim S attributet për të vlerësuar atributet nevojitet një bredhje pasrendore [1].

Përcaktimet me atribute L janë përcaktime që kanë atribute të sintetizuara dhe të trashëguara. Për vlerësimin e tyre përdoren grafet e varësise [1].

### 2.1.17. Mjediset e ekzekutimit

Kodi i burimit i një programi është vetëm një tekst që përmban kodin, instruksione dhe për t'i kthyer në rezultat nevojitet që të kryhen veprimet në makinën e caktuar. Një program ka nevojë për memorie për të ekzekutuar instruksionet. Një program përmban emra për procedurat dhe identifikuesit që duhet të lidhet me memorien aktuale në mjedisin e ekzekutimit. Një mjedis ekzekutimi është një gjendje e makinës e cila përfshin libraritë e softit, ndryshoret e mjedisit,etj. Për sigurimin e shërbimeve që procesojnë programin që është në ekzekutim në sistem. Sistemi i suportit të mjedisit të ekzekutimit i gjeneruar me programin e ekzekutueshem , lehtëson komunikimin ndërmjet proceseve në mjedisin e ekzekutimit. Merret me alokimin dhe dealokimin e hapësirës në memorie gjatë kohës që programi po ekzekutohet[5].

### 2.1.18. Pemet e aktivizimit

Një program është një sekuencë instruksionesh të kombinuara në procedura. Instruksionet në procedura ekzekutohen në menyrë sekuenciale. Çdo gjë që ndodhet ndërmjet fillimit dhe fundit të procedurës quhet trupi i procedurës dhe është i përbërë nga një numër i fundëm instruksionesh . Ekzekutimi i një procedure quhet aktivizim. Një rekord aktivizimi përmban të gjithë informacionin e nevojshëm për thërritjen e një procedure. Një rekord aktivizimi në varësi të gjuhës së programimit të përdorur mund të përmbajë:

**Informacionet që mund të përmbajë një rekord aktivizimi**

|  |  |
| --- | --- |
| Temporaries | Ruan vlerat e ndërmjetme dhe të përkohshme të një shprehje. |
| Local Data | Ruan të dhënat lokale të procedurës së thërritur. |
| Machine Status | Ruan statusin e makinës si regjistrat, program counteri etj. përpara thërrijtes së metodës. |
| Control Link | Ruan adresën e rekordit të aktivizimit të procedurës së thërritur. |
| Access Link | Ruan informacionin e të dhënave që janë jashtë skopit lokal. |
| Parameter Aktual | Ruan parametrat aktuale që përdoren për të dërguar informacionin të procedurave të thërritura. |
| Vlera e kthyer | Ruan vlerat që do të kthejë procedura e thërritur. |

Informcionet jane marre nga nje artikull ne internet ku autori nuk ishte permendur[5].

Kurdoherë që një procedurë ekzekutohet , rekordi i aktivizimit ruhet në stivë, që njihet me ermin si stiva e kontrollit. Kur gjatë ekzekutimit të një procedure thërritet një procedurë tjetër, ekzekutimi i procedurës ndërpritet derisa të përfundojë ekzekutimin procedura e thërritur [5].

Kur procedura e thërritur përfundon ekzekutimin e saj , e transferon kontrollin e saj te procedura që e ka thërritur. Për të përfaqësuar këtë rrjedhë kontrolli është më lehtë duke përdorur formën e pemës, pemët e aktivizimit [5].

### 2.1.19. Shpërndarja (alokimi) e kujtesës

Mjedisi i ekzekutimit menaxhon kërkesat për memorie nga :

* Kodi është programi i shkruar në tekst dhe nuk ndryshon në kohën e ekzekutimit. Kërkesat për memorie janë të njohura në kohën e kompilimit.
* Procedurat pjesa e tekstit është statike por mënyra se si thërriten është mënyrë në të rastësishme. Prandaj përdoren strukturat e të dhënave stiva për të menaxhuar thërritjen e procedurave dhe aktivizimin e tyre.
* Ndryshoret , ndryshoret njihen vetëm në mjedisin e ekzekutimit , përveç rasteve kur ato kanë skop global ose janë konstante. Për të menaxhuar alokimin dhe dealokimin e memories për ndryshoret në kohën e ekzekutimit përdoren strukturat e të dhënave turra(heap).

Në këtë skemë alokimi, të dhënat e kompiluara janë të lidhura me një pozicion fiks në memorie dhe nuk ndryshojnë gjatë gjithë kohës së ekzekutimit të programimit. Duke qenë se kërkesat për memorie dhe vendndodhjet e kujtesës njihen që më përpara , nuk ka nevojë që paketa e suportit të ekzekutimit për alokimin e dealokimin e memories [5].

Thërritja e procedurave dhe aktivizimi i tyre menaxhohen duke përdorur alokimin e memories së stivës. Struktura e stivës është strukturë të dhënash LIFO që e fundit që vjen është e para që ekzekutohet , kjo mënyrë është shumë e dobishme në thërritjen rekursive të procedurave [5].

Alokimi dhe dealokimi i ndryshoreve lokale të procedurës bëhet vetëm në kohën e ekzekutimit. Turrat përdoren për alokimin dinamik të memories për ndryshoret e dealokimin e memories kur ndryshoret nuk janë më të kërkuara. Stivat dhe turrat në ndryshim nga alokimi statik ato mund te rriten ose zvogëlohen në mënyre dinamike dhe të papritur, prandaj nuk mund të caktohet një madhësi fikse memorie për to [5].

### 2.1.20. Tabela e simboleve

Tabela e simboleve është një strukturë të dhënash e rëndësishme e krijuar dhe menaxhuar nga kompilatorët në mënyrë për të ruajtur informacionin për entitete të ndryshme si emra ndryshoresh,emra funksionesh,objektesh,klasash ,ndërfaqjesh etj. Tabela e simboleve përdoret nga të dyja etapat e kompilimit , analiza dhe sinteza [6].

Tabela e simboleve në varësi të gjuhës së programimit mund të përdoret:

* Për të ruajtuar emrat e të gjithë etniteve në një formë të strukturore
* Verifikimin nëse një ndryshore është deklaruar.
* Implementimin e kontrollimin e tipit,verifikimin e vlerëdhënieve dhe shprehjet që janë korrekt nga ana semantikore.
* Percaktimin e skopit të një ndryshore.

Një tabelë simbolesh mund të jetë një tabelë lineare ose tabele hash. Kjo tabelë mban informacione për çdo hyrje në formatin <symbol name,type,attribute>.

N.q.s kompilatori do të menaxhojë një sasi të vogël të dhënash, atëherë tabela e simboleve mund të implementohet si një listë e parenditur, e cila është e lehtë për t’u koduar dhe është e përshtatshme vetëm për tabela që mbajnë sasi të vogëla të dhënash. Një tabelë mund të implementohet me:

* Lista lineare(të radhitura ose të paradhitura)
* Pemët e kërkimit binare
* Tabelat hash

Shumica e kompilatorave përdorin tabelat hash për implementimin e tabelave të simboleve ku vetë kodi burim trajtohet si çelës dhe vlera e kthyer është informacioni rreth simbolit [6].

Insertimi, që përdoret shpesh në fazën e analizes ,shton në tabëlen e simboleve pasi janë identifikuar tokenat dhe emrat e tyre ruhen në tabelen e simboleve.Atributet për një simbol në kodin burim janë atribute që shoqërohen me atë simbol dhe mund të përmbajë vlerë, gjëndje,skop [6].

Lookup është veprimi i kërkimit në tabelën simboleve nëpërmjet emrit për të përcaktuar nëse:

* Ekziston simboli në tabelën simboleve.
* Nëse është deklaruar përpara se të përdoret.
* Nëse emri i ndryshores është perdorur në atë skop.
* Nëse simboli është inicializuar.
* Nëse simboli është deklaruar disa herë.

Formati i funksionit lookup ndryshon në varësi të gjuhës së programimit [6].

### 2.1.21. Menaxhimi i shtrirjes së aksesit të ndryshorëve

Një kompilator përmban dy tipe tabelash simbolesh: një tabelë globale simbolesh e cila mund të aksesohet nga të gjitha procedurat dhe një tabelë skopi simbolesh që krijohen për çdo skop në program. Për të përcaktuar emrin e skopit, tabelat e simboleve organizohen në një strukturë hierarkike[6].

Sa herë që kemi nevojë të kërkojmë nga hierarkia e struktures së të dhënave të tabelës së simboleve nga emri përdorim algoritmin e mëposhtëm:

* Ne fillim simboli do të kërkohet te skopi aktual, në tabelën aktuale të simboleve.
* N.q.s emri gjendet atëherë kërkimi përfundon, përndryshe do të kërkohet tek tabela e simboleve prind.
* Kjo vazhdon derisa të kemi gjetur emrin ose të kemi arritur deri tabela globale e simboleve[6] .

### 2.1.22. Kodi i ndërmjetëm

Disa nga arsyet pse kompilatori duhet të përkthejë kodin burim në kod të ndërmjetëm në vend që ta përkthejë atë direkt në kod makinë:

* N.q.s kompilatori përkthen kodin burim në kodin makinë pa gjeneruar kod të ndërmjetëm, atëherë për çdo makinë të re do të duhej një kompilatori i ri.
* Kodi i ndermjetëm eliminon nevojën për një kompilator të ri për çdo makinë unike duke mbajtur etapën e analizës te njëjtë për të gjithë kompilatorët.
* Etapa e dytë e kompilimit; sinteza, ndryshon në varësi të makinës ku do të përdoret.
* Është më i lehtë aplikimi i modifikimit të kodit burim për të përmirësuar performancën e kodit duke aplikuar teknika optimizimi në kodin e ndërmjetëm [13].

Kodi i ndërmjetëm mund të përfaqësohet në mënyra të ndryshme ku secila mënyrë ka avantazhet e veta:

* Përfaqësimit i nivelit të lartë të kodit të ndërmjetëm është një përfaqësim i kodit që është përafërt me kodin burim. Mund të gjenerohet shumë lehtë nga kodi burim, dhe mund të aplikohen shumë lehtë modifikime për përmirësimin e performancës.
* Përfaqësimi i nivelit të ulët të kodit të ndërmjetëm është një më afër kodit makinë që bën më të përshtatshme për optimizimet që varen nga makina ku ekzekutohen [13].

Gjeneruesi i kodit të ndërmjetëm merr si input pemën e sintaksës me atribute nga faza e analizës semantikore . Pastaj kjo pemë sintaksore konvertohet në një përfaqësim linear , në shprehje passhtresore. Kodi i ndërmjetëm ka tendecën të jetë i pavarur nga makina ku ekzekutohet [13].

### 2.1.23. Gjenerimi i kodit

Gjenerimi i kodit mund të konsiderohet si faza finale e kompilimit, por edhe pas gjenerimit të kodit ka faza të tjera optimizimi të kodit që mund të aplikohen. Kodi i gjeneruar nga kompilatori është vetë kodi objekt i një gjuhe programimi të niveli të ulët , si psh gjuha assembler [14].

Një gjenerues kodit pritet të ketë një njohuri të makinës ku do të ekzekutohet dhe bashkësinë së instruksioneve të asaj makine. Gjeneruesi i kodit të marrë në konsideratë këto gjëra për gjenerimin e kodit:

* Gjuhën e makinës ku do të ekzekutohet, se cfarë arkitekture kanë, procesore CISC dhe RISC .
* Tipin e përfaqësimit të kodit të ndërmjetëm
* Alokimi i regjistrave, gjeneruesi i kodit cakton se çfarë vlera do të mbahen në regjistra dhe në cilat regjistra do të mbahen këto vlera.
* Radhitja e instruksioneve te gjeneruesi i kodit përcakton rradhën në të cilën do të ekzekutohen instruksionet [14].

Gjeneruesi i kodit duhet të gjurmojë regjistrat për disponibilitetin e tyre dhe adresat për vendndodhjen e vlerave gjatë gjenerimin të kodit. Për të bërë këto përdor dy regjistra , përkatësisht Register descriptor dhe Adress Descriptor [14].

* Register Descriptor përdoret për të informuar gjeneruesin e kodit për regjistrat që janë të disponueshëm. Register Descriptor gjurmon vlerat e ruajtura në regjistra dhe kur një regjister i ri duhet gjatë gjenerimit të kodit, konsultohet me descriptorin për disponueshmerine e regjistrit.
* Address Descriptor , vlerat e emrave të identifikuesve në program mund të ruhen në vendndodhje të ndryshme gjatë ekzekutimit. Address Descriptor përdoren për të gjurmuar vendodhjet e memories ku ruhen vlerat e identifikuesve. Këto vendodhje mund të jenë regjistrat e cpu, turrat, stivat, memorja ose nje kombimin i tyre [14].

### 2.1.24. Optimizimi i kodit

Optimizimi i kodit është një teknike transformimi programi që kërkon të përmirësojë kodin duke konsumuar më pak burime . Në optimizimin , programet e ndërtuar në nivelin e lartë zëvëndësohen me kod të gjuhës së nivelit të ulët që është shumë eficente [7].

Një proces optimizimi përfshin tre rregullat e mëposhtme:

* Kodi pas optimizimit nuk duhet të prekë apo të ndryshojë kuptimin e programit.
* Optimizimi duhet të rrisë shpejtësinë e programit dhe nëse është mundur të reduktojë numrin e burimeve që përdor.
* Vetë optimizimin duhet të jetë një proces i shpejtë dhe nuk duhet të vonojë procesin e kompilimit [7].

Përpjekjet për një kod të optimizuar mund të bëhen në disa nivele të procesit të kompilimit.

* Në fillim , përdoruesi mund ta ndryshojnë kodin burim duke perdorur algoritma më eficentë për shkrimin e kodit.
* Pas gjenerimit të kodit të ndërmjetëm , kompilatori mund të modifikojë kodin e ndërmjetëm duke llogaritur adresen dhe permisuar ciklet.
* Gjate prodhimit të kodit makinë , kompilatori mund të përdorë hierarkine e memories dhe regjistrat e CPU në menyrë më eficente [7].

Optimizimi kategorizohet në dy tipe: i pavaruar nga makina ku po ekzekutohet dhe i varur nga makina ku po ekzekutohet.

Optimizimi i pavaur nga makina: kompilatori merr kodin e ndërmjetën dhe transformon një pjesë të kodit që nuk përfshin regjistrat e cpu apo vendndodhjet absolute të memories [7].

Optimizimi i varur nga makina: bëhet pasi ëhtë gjeneruar kodi makinë dhe mundohet të shfrytezojë në maksimum avantazhin e hierarkisë së memories [7].

## 2.2. Dokumentacioni i teknologjive

Për ndërtimin e kompilatorit që përkthen pseudokodin në kod makine kemi përdorur ndihmën e JavaCC [8] .

JavaCC ka një program që gjeneron analizuesin e tokenave. Për gjenerimin e analizuesit të tokenave me ane të JavaCC bëhet duke përcaktuar shprehjet e rregullta për çdo token që duam të identifikojmë. Pastaj gjeneruesi i analizuesit të leksikut do të prodhojë menaxhuesin e tokenave të shkruar në gjuhën java [8].

Përveç gjenerimit të analizuesit të tokenave , JavaCC ofron mundësinë e gjenerimit të parserit . Për të gjeneruar parserin me anë të JavaCC , kërkohet që të përcaktohet gramatika . Pas përcaktimit të gramatikës gjeneruesi i parserave do të prodhojë një parser të shkruar në gjuhën java [8].

JavaCC është një program që kombinon gjeneruesin e analizuesit leksikor me gjeneruesin e parserit. JavaCC transformon shprehjet e rregullta dhe gramatiken në një kompilator. Vetë JavaCC mund të shikohet si një kompilator ku kodi burim është i përbërë nga shprehje të rregullta dhe gramatika e gjuhës dhe rezultati është një kompilator i shkruar në gjuhën java [8] .

Rregullat për përcaktimin e shprehjeve të rregullta në JavaCC janë si mëposhtë:

**Shprehjet e rregullta ne JavaCC**

|  |  |
| --- | --- |
| “b” | “c’ | Përfaqëson gjithë fjalët që janë b ose c |
| “b|c” | Përfaqëson gjithë fjalët që janë të formës b|c |
| (“b”)\* | Pëfaqëson gjithë fjalët që janë të formës bosh,b, bb, …, bbbb…b (n here b) |
| [“b”, “c”, “d” ] | Përfaqëson gjithë fjalët që janë të formës, b ose c ose d |
| [“A”-“Z”, “a”-“z”] | Përfaqëson të gjithë fjalët që kanë simbole nga a deri z dhe A-Z |
| ~[“b”] | Përfaqëson të gjithë fjalët që nuk kanë simbolin b si pjesë të tyre. |
| ~ “b” | Bie ndesh me rregullat e JavaCC për specifikimin e shprehjeve të rregullta. |
| ~([“b”]) | Bie ndesh me rregullat e JavaCC për specifikimin e shprehjeve te rregullta. |
| ~(“b”) | Bie ndesh me rregullat e JavaCC për specifikimin e shprehjeve te rregullta. |
| ([“0”-“9”])+ | Përfaqëson të gjithë fjalët që kanë vetëm numra : 1,12,…,234… |

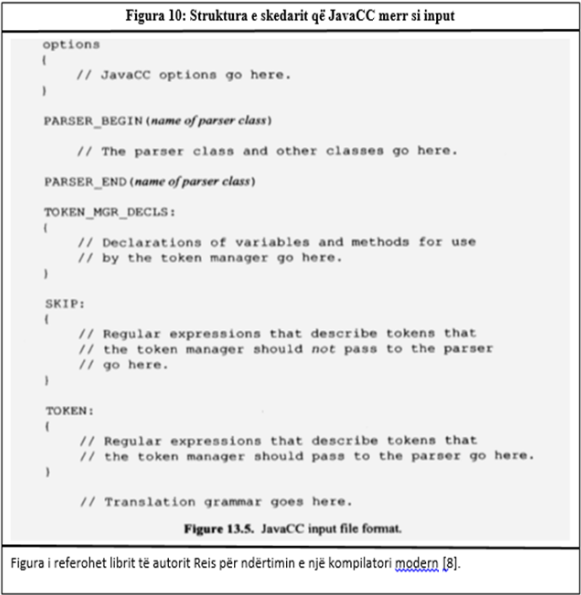
Informacioni i tabeles i referohet tutorialit të JavaCC [8].

JavaCC na lejon që shprehjet e rregullta që ne përcaktojmë të shoqërohen me emra. Për ta bërë këtë ne mbështjellim emrin së bashku me shprehjet e rregullta brenda simbolve < dhe >.

<UNSIGNED: ([“0”-“9”]+)>

<ID: [ "A"-"Z", "a"-"z"] ([ "A"-"Z", " a " - " z " , "0"-"9"] )\*>

Sipas konvecionit , ne perdorin germa kapitale për emërtimin e shprehjeve të rregullta. Me emertimin e shprehjeve të rregullta ne kemi krijuar kategori tokenash që do i referohemi me atë emertim [8]. Skedari që merr si input JavaCC mbaron me mbaresen .jj dhe ka strukturën si mëposhtë:



Në skedarin që merr si input JavaCC lejohen komentet, dhe janë si në gjuhen java , duke përdorur simbolet //komentet dhe /\* komentet \*/ [8].

Blloku options përdoret për të specifikuar opsionet e JavaCC.

P.SH STATIC=false;

Në shembullin e mësipërm opsioni STATIC caktohet false. Duke specifikuar opsionin STATIC si false , bën që JavaCC të gjenerojë metoda statike për gramatikën që i kalohet. Të gjitha opsione e JavaCC kanë vlera default dhe vlera default e opsionit STATIC është true [8].

Ndërmjet PARSER\_BEGIN dhe PARSER\_END shkruhet i gjithë kodi java për programin e JavaCC. Për këtë bllok është e detyrueshme të ketë të paktën një klasë të parserit të javës. Klasa e parserit është nevojshme sepse është kjo klasë në javacc ku vendoset kodit java që do të prodhohet kur përkthehet gramatika [8].

Klasa e parserit duhet specifikuar brenda kllapave të PARSER\_BEGIN dhe PARSER\_END. Nëse parseri është Test atëhere klasa duhet të ndërfutet ndërmjet PARSER\_BEGIN DHE PARSER\_END si mëposhtë:

PARSER\_BEGIN(Test)

…

PARSER\_END(Test)

Nuk është e thënë që klasa e parserit të përmbajë kod java, ajo mund të jetë edhe bosh si mëposhtë:

PARSER\_BEGIN(Test)

public class Test

{

}

PARSER\_END(Test)

Në këtë rast klasa Test do të përmbajë vetëm kod java të prodhuar nga JavaCC gjatë përkthimit të gramatikës. Në mënyrë alternative mund të përfshijmë metoda si main në klasën e parserit [8].

PARSER\_BEGIN(Test)

c l a s s Test

{

public static void main(String[ ] args)

{

}

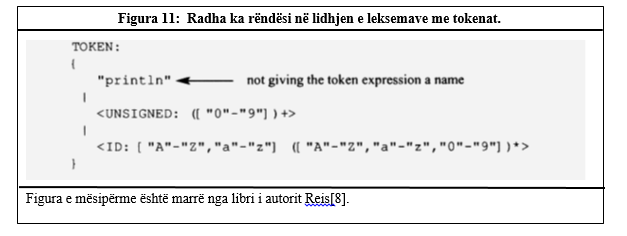
}

PARSER\_END(Test)

Në këtë rast klasa Test do të përmbajë metodën main sëbashku me metodën e gjeneruara gjatë përkthimit të gramatikës [8].

Blloku i TOKEN\_MGR\_DECLS përmban skedarët për deklarimet e ndryshoreve dhe metodave që do të përdoren nga menaxhuesi i tokenave (analizuesi i leksikut) [8].

Blloku i tokenave përmban shprehjet e rregullta për çdo token të token menaxherit që do t'i kalohen parserit. Për shkak të emërtimin të shprehjeve të rregullta ne mund të përdorin këto emra në vend të shprehjeve të rregullta për të përfaqësuar tokenat te gramatika. Radha se si listohen shprehjet e rregullta ka rëndësi. N.q.s se dy shprehje të rregullta që përputhen me të njëjtën lekseme, atëhere do të zgjidhet ajo shprehje e rregullt që është vendosur më përpara në bllokun e tokenave. Leksema println përputhet edhe me shprehjen rregullt të tokenit PRINTLN dhe te tokenit ID, por n.q.s PRINTLN është vendosur përpara ID në bllokun e tokenave atëhere menaxhuesi i tokenave do të trajtojë atë lekseme si token të kategorisë PRINTLN [8].



Blloku SKIP përdoret për të specifikuar shprehjet e rregullta për simbolet që nuk do të merren pararsysh nga kompilatori. Këto shprehje të rregullta që përcaktohen në bllokun SKIP do të anashkalohen në fazën e analizës leksikore [8].



Specifikimi i parserit bëhet duke specifikuar prodhimet e gramatikës. Prodhimet e gramatikës specifikojnë sekuencat e tokenave që janë të pranueshme nga parseri [9].

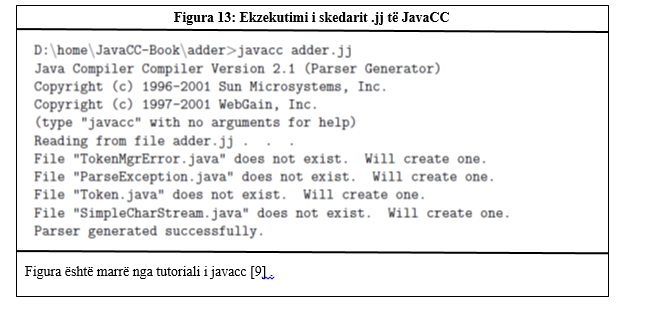
Shembulli i mëposhtëm duket si dekalarimi i një metodë në java. Prodhimi i mëposhtëm që kemi marrë si shembull që përcakton një sekuencë me tokenat që fillojnë tokenin NUMBER dhe mbarojnë me tokenin EOF dhe ndërmjet tyre mund të jenë zero ose me shumë sekuencash të tokenit PLUS që ndiqet nga tokeni NUMBER. Parseri që kemi specifikuar më poshtë do të kontrollojë nëse sekuenca përputhet me rregullat e gramatike ,nuk do të kryejë veprimin e mbledhjes së dy numrave [9] .

void Start() :

{}

{ <NUMBER> ( <PLUS> <NUMBER> )\* <EOF> }

Gjenerimi i parserit dhe analizuesit të leksikut bëhet duke ekzekutuar në CMD komandën javacc mbi skearin adder.jj te javacc [9].



Pasi ekzekutojmë skedarin adder.jj me komandën javacc, gjenerohen 7 klasa java [9].

TokenMgrError është një klase që përdoret për të detektuar gabimet gjatë fazës së analizës leksikore [9].

ParseException është një klase që përdoret për të detektuar gabimet gjatë fazës së parsimit [9].

Token është një klase që përfaqëson tokenat. Çdo objekt token nga një fushë të tipit integer që quhet kind dhe përfaqëson kategorinë e tokenit, dhe një fushë string që emërtohet image dhe përfaqëson sekuencën karaktereve të ardhura nga skedari input që përfaqësojnë tokenat [9].

SimpleCharStream është një përshtatës që dërgon karekteret te analizuesi leksikor [9].

AdderConstants është një ndërfajqe që përcakton numrin e klasave që përdoren në analizën leksikor dhe në analizën sintaksore [9].

AdderTokenMangaer është analizuesi leksikor [9].

Adder është parseri [9].

Pasi kemi gjeneruar të gjithë klasat me javacc , atëhere ne mund të ekzekutojmë skedarët e prodhuar në java me komanden javac \*.java [9].

Nëse i japin si input sekuencën e e karaktereve “123-456\n”, do të gjenerojë gabim leksikor sepse ka një karakter që nuk është specifikuar siç është simboli minus. Në këtë rast programi do të hedhë si përjashtim klasën TokenMgrError [9].

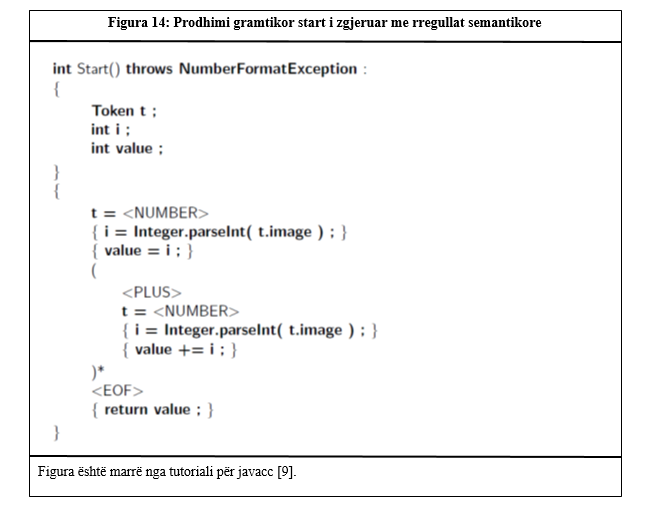
Nëse i japin si input sekuencën e karaktereve “123++456\n” , do të gjenerojë gabim sintaksor, sepse sekuenca e tokenave nuk përputhet me specifikimin e gramatikës.Në këte rast programi do të hedhë gabim përjashtimin ParseException [9].

Ne si japing si input sekuences e karaktereve “1+2\n” atëhere programi nuk do të hedhë asnjë përjashtim dhe do të përfundojë së ekzekutuari. Ky është rasti kur sekuenca e tokenave përputhet me specifikimin e gramtikës [9].

Metodat e gjeneruara nga javaCC për prodhimet e gramatikës thjesht kontrollojnë nëse input që i kalohet skedarit përputhet me specifikimet e gramatikës dhe me specifikimet e tokenave. Për të zgjeruar parserin që n.q.s inputi përputhet me specifikimet e gramatikës të bëjë një veprim të caktuar ne mund të shtojmë kod në java në prodhimet gramatikore [9] .

Në shembullin e mëposhtëm kemi bërë rregullat e gramatikës të shoqërohen me veprime në rast se përputhen me emrin e inputit të sekuencës së tokenave [9].

Tipi i kthimit ndryshohet nga void në int për prodhimin BNF. Kemi deklaruar përjashtimin NumberFormatException për metodën gjeneruar nga prodhimi gramtikor. Ne kemi deklaruar 3 ndryshore, ku ndryshorja t është e tipit token, dhe fusha image e klasës së tokenave ruan vargun me karaktere që përputhet me tokenin. Kur një token përputhet në një prodhim të BNF , ne mund të regjistrojmë objektin token si referencë e klasës token [9].



Në shembullin e mesipërm kur kalojmë një sekuencë tokenash që përputhet me gramatikën e specifikuar tek metoda start atëhere përveç se vargu me karaktere do të njihet nga programi ai do te japë rezultatin që do të vijë nga shuma e numrave që i kalohet [9].

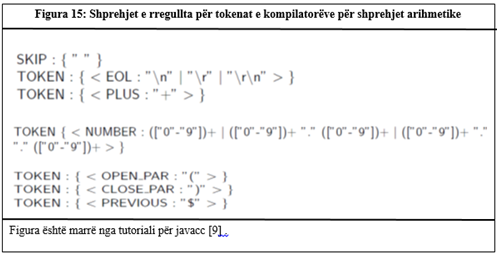
t=<NUMBER> {i=Integer.parseInt(t.image);} Këto dy rreshta marrim vlerën e tokenin , dhe më pas e konvertojnë në numer që të bëjnë veprimin e mbledhjes [9].

## 2.3. Punë të tjera në lidhje me këtë teme

### 2.3.1. Kompilator për shprehjet aritmetike

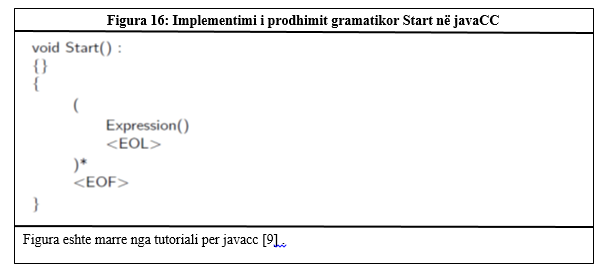
Kompilator për shprehjet aritmetike është një kompilator i thjeshtë , që merr si input shprehjet ari]tmetike dhe kthen rezultatin e shprehjes [9].

Në fillim të ndërtimit të kompilatorit fillojmë të specifikojmë tokenat duke përcaktuar emrat e tokenave të shoqëruara me shprehjet e rregullta . Nga specifikimi i shprehjeve të rregullta për tokenat , javacc gjeneron analizuesin leksikor [9].

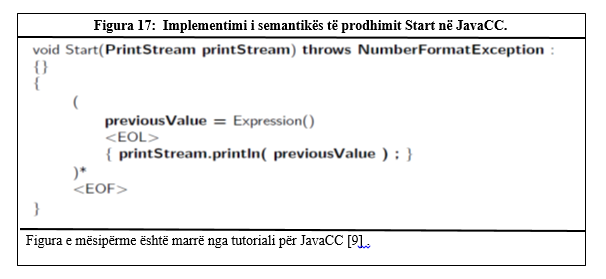


Inputi që i kalohet parserit është një sekuencë rreshtat që përbajnë shprehjen e aritmetikore [9].

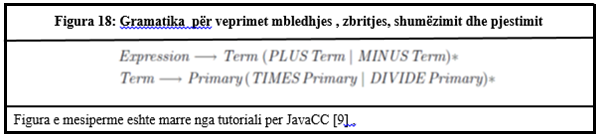
Prodhimi gramatikor Start🡪(Expression EOL)\* EOF shkruhet si mëposhtë në javacc për të gjeneruar parsererin [9].



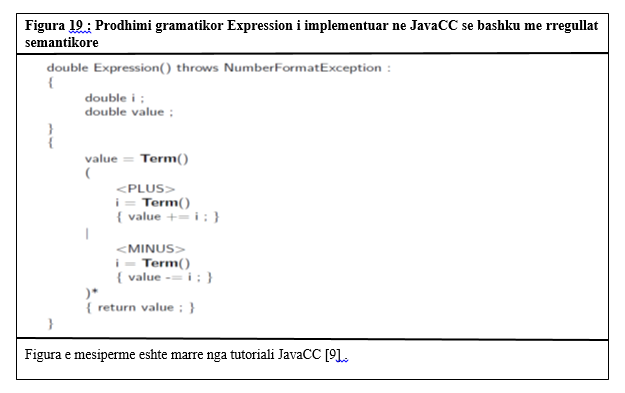
Më sipër kemi specifikuar thjesht parserin, nuk kemi shtuar ndonjë veprim që të kryhet në rast se sekuenca e tokenave përputhet me specifikat e gramatikës [9].



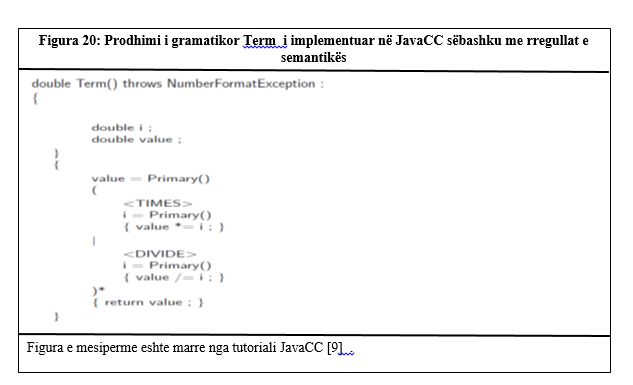
Në rastin e mësipërm prodhimeve gramatikore i kemi shtuar veprime në rastin kur sekuenca e tokenave përputhet me specifikimin e gramatikës. Në këte rast veprimi që kryhet është afishimi i vlerës që kthehet nga prodhimi gramtikor expression. Prodhimi i expression është përgjegjës për parsimin e shprehjeve aritmetikore, si dhe për kthimin e vlerës që do të kthehet pas llogaritjes së shprehjes aritmetikore [9].



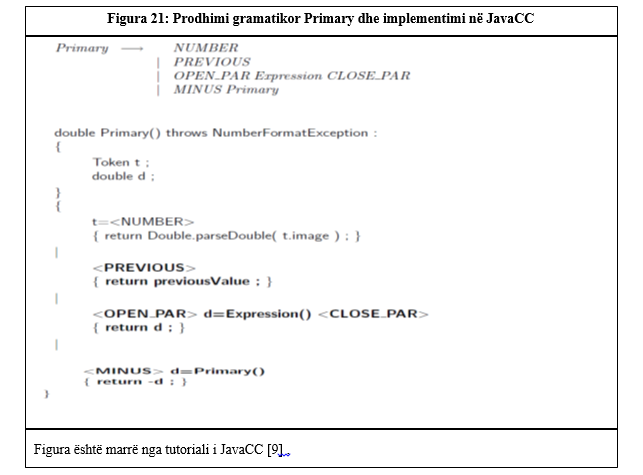
Prodhimet e gramatikës së mësipërme janë bërë duke respektuar prioritetin e operatoreve aritmetikë. Që shumëzimi dhe pjesëtimi vlerësohen përpara mbledhjes dhe zbritjes, përveç rastit kur janë brenda kllapave [9] .



Vlera që kthehet nga metoda e prodhimit Expression të gramatikës së gjuhës është e barabartë me mbledhjen ose zbritjen e vlerës që kthehet nga prodhimi i Term. Në këtë mënyrë do të bëhet veprimi i prodhimit i Term dhe pasi është kthyer vlera pas vlërësimit të prodhimit Term do të bëhet mbledhja ose zbritja [9] .



Prodhimi Term kthen rezultatin që vjen si shumëzim apo pjesëtim i vlerave të kthyera nga prodhimi Primary. Përpara se të kryhet veprimi i shumëzimit apo pjesëtimit, vlerësohet prodhimi Primary. Në këtë mënyrë ne mund të bëjmë që të vlerësojmë shprehjet brenda kllapave përpara veprimit të shumëzimit dhe zbritjes [9].



Prodhimi Primary kthen vlerën e një numri të një shprehje aritmetikore brenda kllapave ose një numër negativ [9].

Ky është një shembull shumë i thjeshtë i një kompilatori, por shumë i dobishëm në kuptimin e pjesëve bazë në ndërtimin e një kompilatori. Metoda e llogaritjes së rezultatit në këtë punë është metoda direkte, ku kodi java është nderfutur brenda prodhimeve gramatikore. Kjo menyrë e të punuarit në ndërtimin e kompilatorëve në javacc mund të jetë në rregull për kompilatorë të vegjël por nuk mund të përdoret në kompilatorë më kompleksë.Për këto është nevojshme ndarja e logjikës nga analiza sintaksorë [9].

### 2.3.2. Kompilator për aksesimin e databazës së një videoteke

Supozojmë se kemi një databazë më të dhëna për filmat, aktoret dhe fjalet kyçe . Tabela e filmave mban të dhënat për filmin si titulli,regjizori etj. Tabela e aktoreve mban listën me aktorë për çdo film. Fjalët kyçe përmbajnë fjalë që përshkruajnë filmin si aksion, drama, aventure [10] .

Klientet do të donin të kishin mundësi të kërkonin këtë databazë për filmat që dëshironin duke futur infomacionin si më poshtë:

actor = "Christopher Reeve" and keyword=action and keyword=adventure

dhe do të donin t'i kthehej përgjigje filmi Superman me actor Christopher Reeve.

Gjithashtu klienti do të donte që të përdorte kllapat për të ndryshuar radhën e vlerësimit të informacionit si mëposhtë :

(actor = "Christopher Reeve" and keyword=action) or keyword=romance

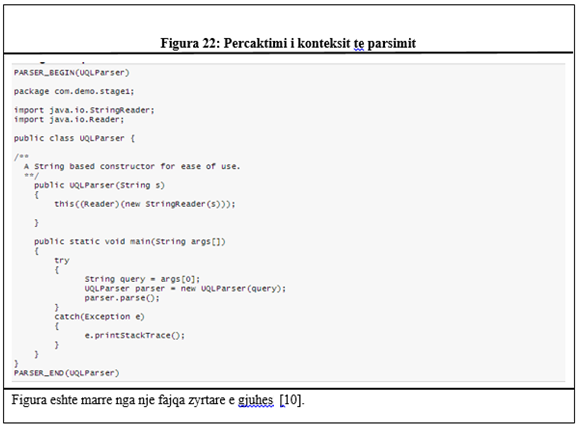
që mund të sjellë si rezultat filma ku nuk merr pjesë aktori Christopher Reeve 10].

Për ta ndërtuar këtë gjuhë sipas kërkesave të klientit duhet te përcaktojmë dy gjera kryesore:

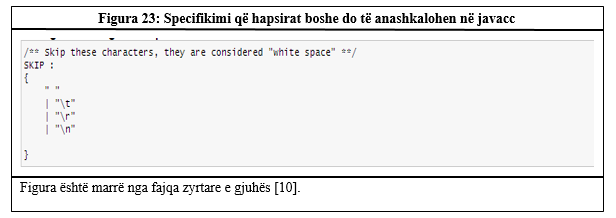
* Përcaktimi i gjuhës në javacc duke siguruar që inputi i përdoruesit parsohet në mënyre korrekte
* Shtimi i veprimeve që shoqërohen me prodhimet gramatikore [10].

Gjuha do të përcaktohet në skedarin e quajtur UQLParser.jj. Ky skedar do të kompilohet nga javaCC në një bashkësi skedarësh të tipit java. Për përcaktimin e gjuhës së skedar .jj te javaC duhet të:

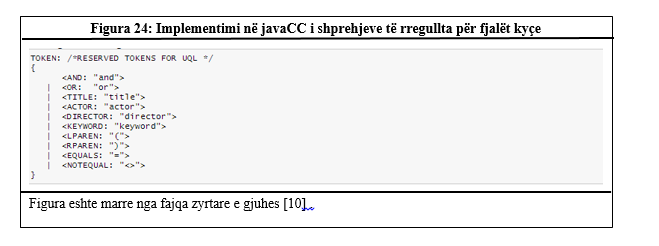
* Përcaktojmi i kontekstit të parserimit.
* Përcaktimi i hapësirave boshe.
* Përcaktimi i tokenave.
* Përcaktimi i sintaksës së gjuhës .
* Përcaktimi i veprimeve të shoqëruara me sintaksën e gjuhës [10].



Në këtë gjuhë ne duam që të anashkalojmë të gjitha hapësirat boshe, tab, rresht të ri etj.



Tokenat që kjo gjuhë do të njohë janë actor = “Christopher Reeve” . Në këtë gjuhë aktori dhe shenja = do të jenë fjalë të rezervuara të gjuhës ashtu siç do të jenë titulli i filmit , kllapat e rrumbullakëta,simbolet e krahasimit etj.. Të gjithë fjalët e rezervuara të gjuhës janë si më poshtë:

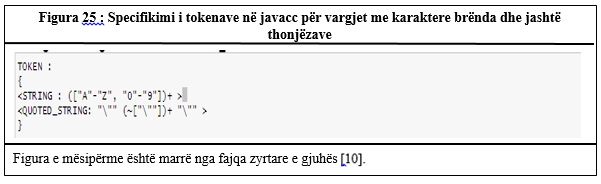


Për tokenin që do të jetë vlera me të cilat do të krahasohet me aktoret, regjizorët ejt, do të përdorim tokenin e mëposhtëm. Shprehja e rregullt e tokenit String i perfshin të gjithë tokenat e mësipërm , prandaj ai nuk duhet të vendosë në fillim, ai duhet të vendoset në fund që përfaqësojë të gjithë fjalët që nuk përkojnë me fjalët e rezervuara [10] .

<STRING : (["A"-"Z", "a"-"z", "0"-"9"])+ >

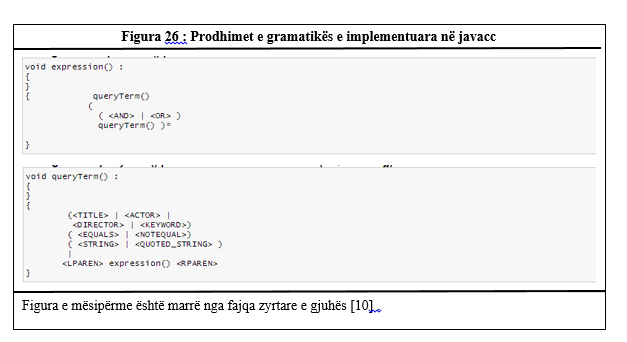
Më siper kemi shprehjen e rregullt për fjalët string që do të përfshijnë kombinime të ndryshme të gërmave të vogla, kapitale dhe numrave[10].

Fjalët brenda thonjëzave dyshe janë më të vështira për të përcaktuar shprehjen e tyre të rregullta. Ne i përcaktojmë vargjet me karaktere brenda thonjëzave dyshe si bashkësi me simbole te ndërfutura brenda thonjëzave dyshe [10].

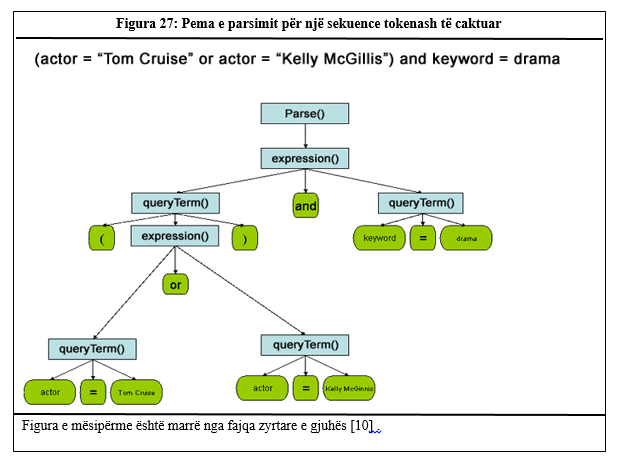


Pasi kemi përcaktuar tokenat e gjuhës tani do të përcaktojmë rregullat e parsimit që do të specifikojnë sekuencen e tokenave që përputhet me gramatikën e gjuhës [10].

Prodhimi queryTerm specifikon një kriter të vetëm , p.sh si title=”Matrix” etj. Një shprehje është ndërfutur brenda kllapave. Prodhimi queryTerm mund të japim është një bashkësi kriteresh brenda kllapave [10].



Pema e parsimit për sekuencën e tokenave të mëposhtme:

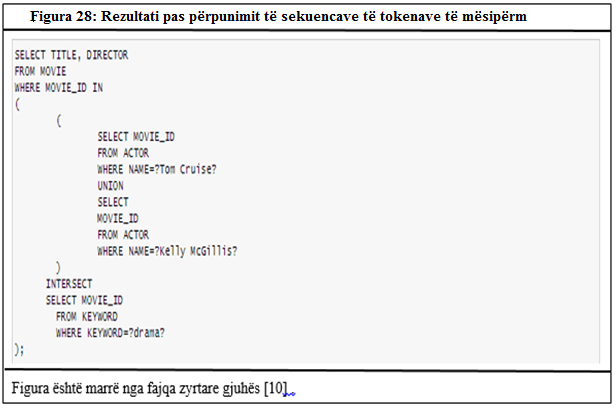


Tani kemi arritur të ndërtojmë një gjuhë që njeh nëse sekuenca e tokenave është pjesë e gjuhës apo jo, por nuk kemi arritur që në rastin kur këto sekuenca tokenash janë pjesë gjuhës të gjenerojnë një rezultat . Tani qëllimi është të kthejmë informacionin nga databaza në varësi të kritereve që marrim nga klienti [10].

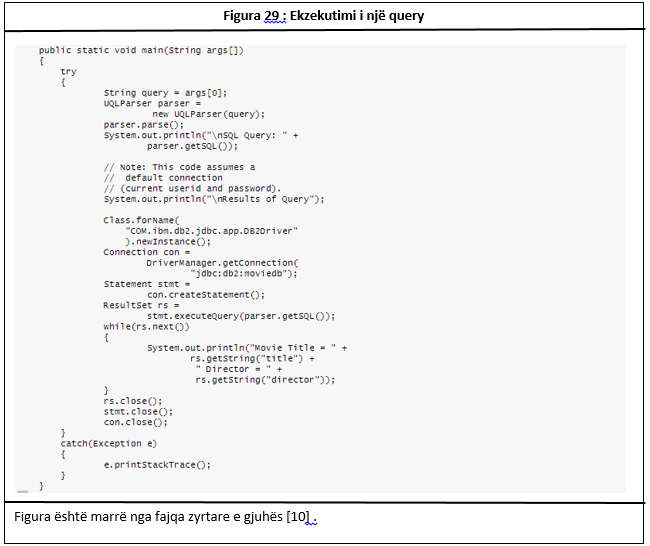
P.Sh sekuenca e tokenave të gjuhës :

(actor = "Tom Cruise" or actor = "Kelly McGillis") and keyword=drama

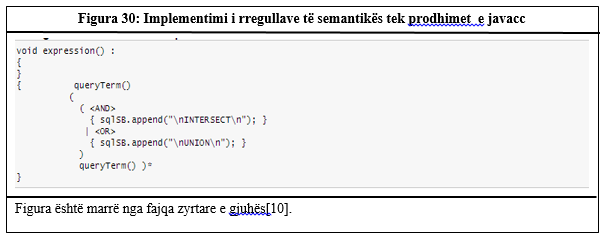
do të përkthehej si mëposhtë në një query sql:



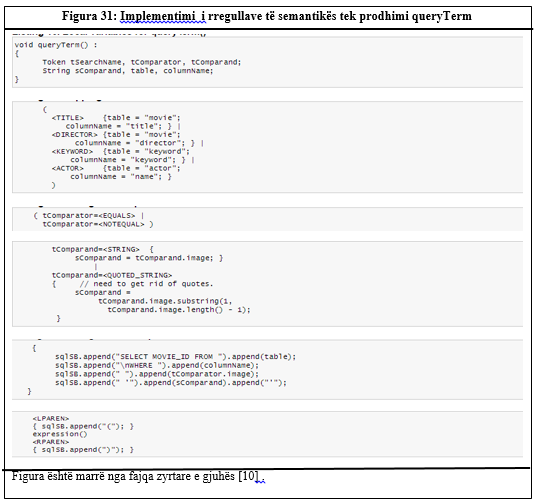
Për të ekzekutuar një query , gjëja e parë që duhet të bëjmë është të krijojmë një lidhje me databazën dhe të krijojmë objektet për të ekzekutuar query [10].



Veprimet që do të lidhen me prodhimin gramatikor expression janë si mëposhtë:



Për prodhimet queryTerm do të na duhen ndryshoret për të marrë informacionin që na nevojitet për të vendosur kriteret tek query [10].



Pastaj ndryshoret lokale qe i deklaruam më siper dhe i lidhem me tokenat prej te cilave do të marrin informacionet që na duhen për të prodhuar rezultat [10].

# 3. Modelimi i zgjidhjes

## 3.1. Konteksti i biznesit

Studentët e vitit të parë mësojnë të shkruajnë algoritmet në pseudokod. Pseudokod është një gjuhë që përmban të gjithë elementet bazë të një gjuhe programimi . Pseudokodi është më i lehtë për të shpjeguar se si funksion logjika e programimit pasi është në gjuhën amëtare të studentëve dhe është i thjeshtë për t’u mësuar, se nuk ka elemente komplekse që mund të kenë gjuhët e programimit. Qëllimi i përdorimit të pseudokodit ështe lehtesimi i mësimit të logjikës së programimit për studentët e vitit të parë. Duke qënë se termat që do të shpjegohen janë në shqip, ata e kanë më të lehtë për të kuptuar konceptet. I vetmi problem është që pseudokodi është nje gjuhë që përdoret për të shpjeguar , dhe nuk përdoret për të zhvilluar programe dhe nuk ka një kompilator që pseudokodi të përkthehet në rezultat. Nëse do të ndërtohej një kompilator për të përkthyer pseudokodin në rezultat do t’i ndihmonte studentet të kuptonin më mire konceptet e programimt që përveç se do të ishin në shqip , do të gjeneroheshin edhe rezultate prej tyre.

Përveç ndërtimit të kompilatorit , duhet edhe një editor që të lëhtësojë punën e shkrimit të pseudokodit, dhe të tregojë gabimet e bera në zhvillimin e kodit.

## 3.2. Rastet e përdorimit/kërkesat

Kompilatori duhet të përmbajë funksionet bazë për leximin e të dhënave nga përdoruesi dhe afishimin e të dhënave nga përdoruesi.

Kompilatori duhet të vlerësojë shprehjen numerike dhe vlerën e tyre duhet ta ruaj në një ndryshore.

Kompilatori duhet të vlerësojë shprehje logjike dhe duhet të ruajë vlerën e tyre në një ndryshore.

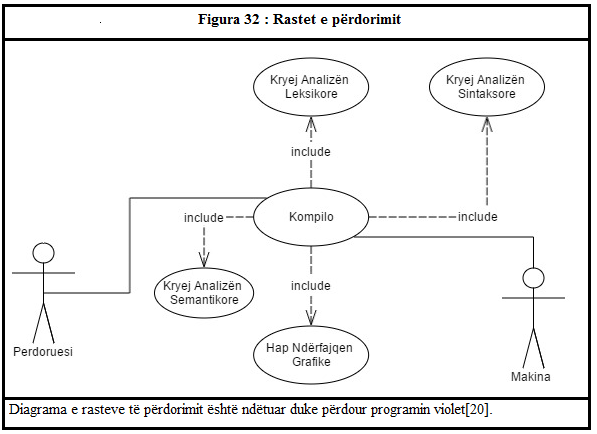
Nuk ka nevojë të përcaktohet tipi që të deklarohen ndryshoret.

Kompilatori duhet të implementojë strukturat e kontrollit të rrjedhës së ekzekutimit.

Duhet të implementojë strukturat e përzgjedhese që në bazë të vlerës së kushtit që është përcaktuar te këto struktura vendoset nëse një instruksion do të ekzekutohet apo ajo.

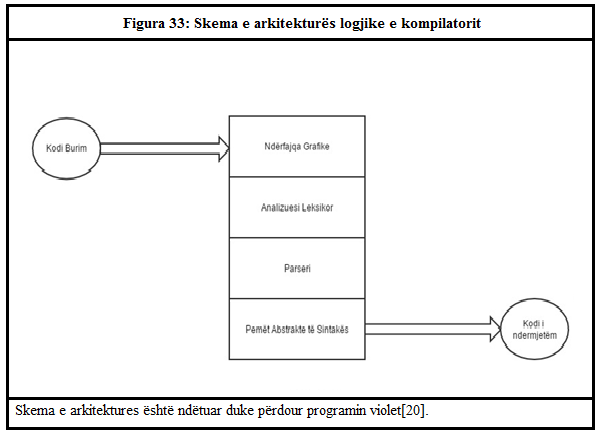
Duhet të implementojë struktura përsëritëse që në bazë të vlerës së kushtit që është përcaktuar vendosim se sa herë do të përsëritet në një instruksion.

Kompilatori duhet të ketë sintaksën sa më të thjeshtë dhe nuk duhet të kerkojë shumë kohe për të ekzekutuar.



## 3.3. Arkitektura Logjike

Arkitekura logjike e kompilatorit për pseudokod është përbërë nga ndërfaqja grafike,tabela e simboleve, analizuesi leksikor,parseri dhe pemët absrakte të sintaksës.



Ndërfaqja grafike është përgjegjëse për marrjen e inputit nga përdoruesi. Ndërfaqja grafike është ajo që shikon përdoruesi, ka një editor ku shkruhet kodi dhe ka opsione të ndryshme për të ruajtur , hapur dhe modifikuar kodin e shkruar.

Analizuesi leksikor është përgjegjës për inputin që merr nga ndërfajqa grafika ta përkthejë në sekuencë tokanash.

Parseri është përgjegjës për transformimin e sekuencës së tokenave të marrë nga analizuesi leksikor në pemë parsimi. Parseri ështe përgjegjës për të përcaktuar nëse një sekuencë tokenash është pjesë e gjuhës dhe n.q.s është ajo ndërton pemën e parsimit.

Pemët abstrakte të sintaksës janë përgjegjëse për shoqërimin e prodhimeve gramatikore me veprime specifike. Ato janë përgjegjëse për gjenerimin e rezultatit kur programi është shkruar me kuptim nga ana semantikore.

# 4. Zhvillimi

## 4.1. Teknologjite e përdorura

Javacc është gjenerues analizuesi leksikor, gjenerues parseri dhe është mjeti kryesore për ndërtimin e kompilatorit. Javacc është open source. Në vitin 1996 kompania sun microsystem nxori një gjenerues parserash të quajtuar Jack. Zhvilluesit që krijuan Jack, ndërtuan kompaninë e tyre të quajtur Metamata dhe ndryshuan emrin nga Jack në Javacc. MetaMata u bë pjesë e WebGain dhe pasi WebGain u mbyll , JavaCC u zhvendos aty ku ndodhet sot. Ka license BSD që lejon të përdoret pa pagesë [15].

Java është gjuha ku është ndërtuar kompilatori. Versioni që është përdorur për këtë detyrë është 1.7 dhe është pronë e kompanisë ORACLE [16].

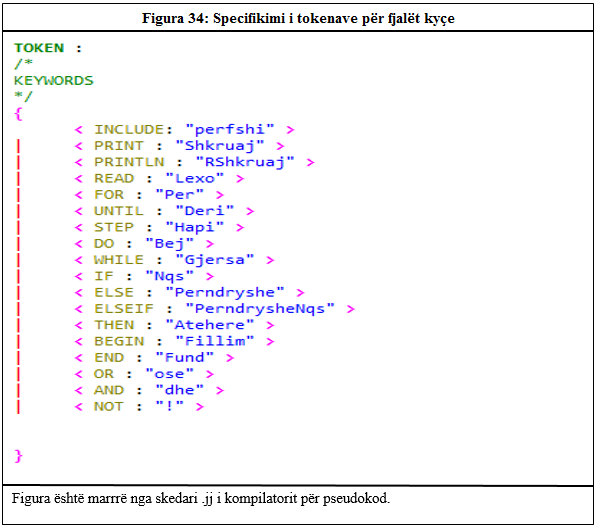
Eclipse Mars është një mjet që ndihmon në zhvillimin e programeve kompjuterik të gjuhëve të ndryshme, ku më shumë është përdorur për java, për opsionet e shumta që ofron specifikisht për këtë gjuhë. Eclipse është ndërtuar në gjuhën java dhe është përdorur kryesisht për zhvillimin e aplikacioneve java. Versioni Eclipse që është përdorur tek kjo detyrë është Mars 4.5. Ka licensë Eclipse Public Licesen [17].

JavaCC Eclipse Plugin është një modul që integrohet te editori eclipse dhe ndihmon në gjenerimin e klasave që derivojnë nga skedari .jj i javacc . Versionet me të cilat mund të integrohet ky plugin është me Mars(4.5),Luna(4.4), Kepler(4.3), Juno(4.2,3.8). Ky plugin ka licensë GPL që lejon të përdoret pa paguar ndonjë tarifë [18].

## 4.2. Detaje mbi zhvillimin

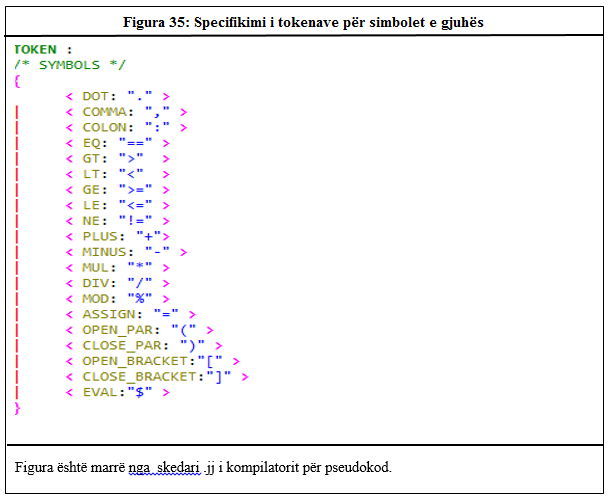
Në fillim kemi përcaktuar analizuesin leksikor. Javacc ofron gjenerimin e analizuesit leksikor nëse përcaktohen shprehjet e rregullta dhe emri i tokenit që përputhet me shprehjet e rregullta.

Kemi specifikuar fjalët kyçe të gjuhës ,që janë pjesë e gjuhës dhe nuk mund të përdoren si identifikues.



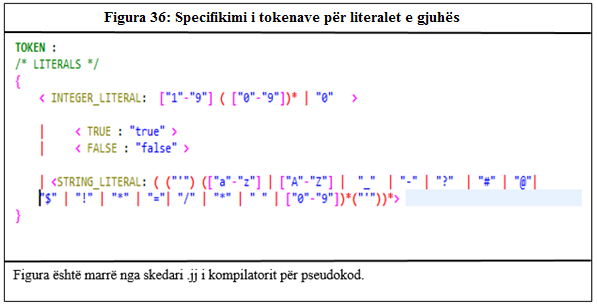
Përcaktimi i shprehjes së rregullt për çdo fjalë kyçe të gjuhës ishte e thjeshtë, pasi që një sekuencë simbolesh të jenë pjesë e një tokeni që specifikon fjalët kyçe duhet patjeter të përputhet me karakteret që përbëjnë fjalët kyçe, gërmat e vogla dhe kapitale kanë rëndësi në përcaktimin nëse një sekucencë simbolesh është pjesë e tokenit që specifikon fjalët kyçe.

Përcaktimi i shprehjeve të rregullta për simbolet të gjuhës është akoma më i thjeshtë se përcaktimi i fjalëve kyçe. Për çdo simbol përcaktojmë emrin e tokenit dhe shprehjen e rregullt. Shprehja e rregullta është vetëm duke specifikuar karakterin e simbolit që do të indentifikohen të gjithë me atë token nëse përputhen me shprehjen e rregullt.



Literalet janë vlera fikse që marrin ndryshoret. Në gjuhën e pseudokodit ne kemi vetëm 3 tipe literalesh që mund të marrë një ndryshore. Ato janë:

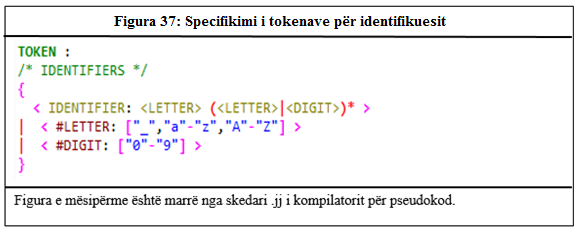
* Numra të plotë
* Vlera logjike (booleane) 0 ose 1
* Stringje (Varg karakteresh)

Shprehja e rregullt që pëcakton tokenat e numrave të plotë përputhet me të gjithë ato sekuenca karakteresh qe janë numrat. Numri më i vogël që mund të pranohet është 0 dhe teorikisht nuk ka limit për numrin më të madh që mund të pranohet, kufizimi mund te vije nga hapsira në memorie.

Shprehja e rregullt që përcakton tokenat e vlerave logjike është njësoj si fjalët kyçe. Vlerat logjike kanë vetëm dy gjendje e vertete apo e gabuar prandaj kemi dhe dy tokena. Tokeni për vlerën logjike kur është true dhe tokeni për vlerën logjike kur është false.

Shprehja e rregullt për vargun me karaktere është më komplikuara . Në gjuhën e pseudokodit , vargjet me karaktere janë të ndërfutura brenda thonjëzave njëshe. Çdo karakter i ndërfutur brënda thonjëzave njëshe njihet si tokeni të vargut të karaktereve.

Identifikuesit përdoren për emërtimin e ndryshoreve ,funksioneve ose tipeve te reja . Kompilatori i ndërtuar për pseudokod nuk i ka akoma të implementuara funksionet dhe tipi i vlerës , ndryshoret e përcaktojnë në fazën e inicializimit. Identifikuesit për këtë kompilator janë përdorur vetëm për emërtimin e ndryshoreve.

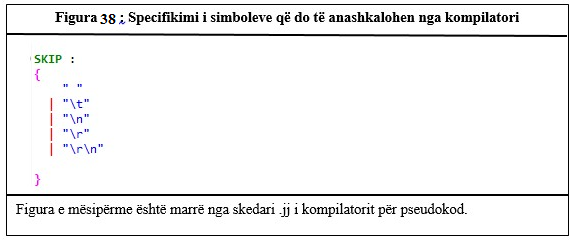


Të gjithë sekuencat e karaktereve që plotësojnë kushtet e mëposhtëme:

* Fillojnë me një gërmë vogël, kapitale ose simboli \_ .
* Mund të ndiqen nga gërmë e vogël, kapitale ose numra ose simboli i \_
* Nuk përkojnë me shprehjet e rregullta të fjalëve kyçe

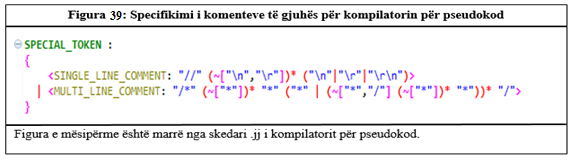
atëhere bëjnë pjesë te tokeni i identifikuesve.

Në javacc të lejojnë të specifikosh karaktere që nuk janë të rëndësishme. Këto karaktere flaken tutje nga analizuesit leksikor. Të gjitha sekuencat e karaktereve që përkojnë me shprehjet e rregullta të specifikuara tek opsioni skip do të flaken tutje nga analizuesi leksikor.



Më sipër kemi specifikuar që të gjithë karakteret boshe të flaken tutje dhe të mos kenë ndikim në percakimin e tokenave. Kjo bëhet për arsye që numri i hapësirave boshe të mos ketë ndikim në gjuhën e pseudokodit. Që kodi mund të shkruhet edhe në një rresht të vetëm ose në shumë rreshta dhe do të pranohej nga gjuha nëse do të ishte shkruar saktë.

Nuk ka gjuhë programimi pa komente. Çdo gjuhë programimi ka një mënyrë se si mund të vendosësh komente. Komentet përdoret për të shpjeguar pjesë të kodit për ta pasur më të lehtë për t’ju kthyer kodit për modifikime apo për t’ia shpjeguar dikujt tjetër. Komentet injorohen nga parseri por njihen nga analizuesi leksikor. Prandaj komentet futen tek opsioni SPECIAL TOKEN që përcakton tokena që nuk marrim pjesë në procesin e parsimit.



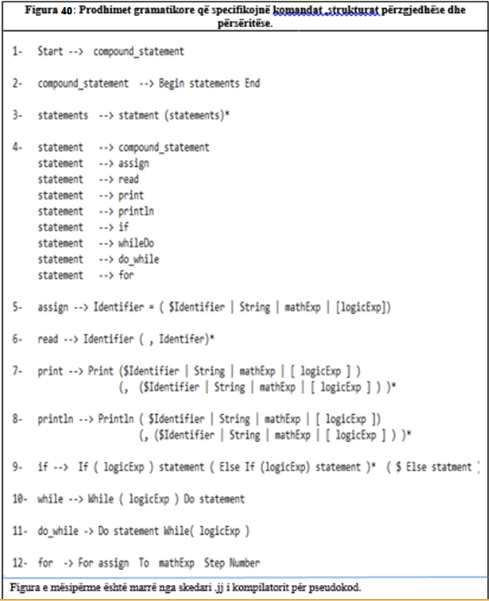
Analizuesi leksikor i kompilatorit për pseudokod njeh dy tipe komentesh. Komentet ndryshojnë nga njeri tjetri te numri i rreshtave që përcaktojnë.

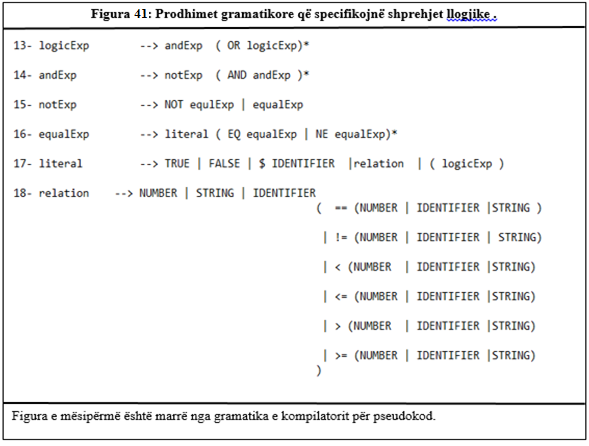
Kemi komentet që janë të përcaktur brënda një rreshti. Çdo karakter pas dy simboleve / dhe brënda rreshtit ku është përcaktuar njihet si token i komentit nje rreshtor.

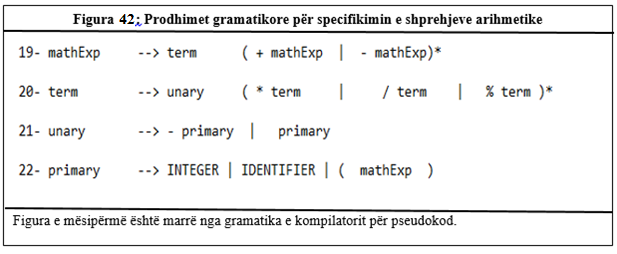
Komentet shumë rreshtorë përcaktojnë si token të gjithë karakteret që janë të ndërfutura brenda simboleve /\* \*/. Këto komente mund të jenë nga një rresht në shumë rreshta.

Përpara se të gjenerojmë prodhimet në javacc, ne kemi ndertuar një gramatike që është LL(1). Parseri që gjeneron JavaCC është një parser top down dhe gramatikat që njeh ky parser janë gramatikat LL1.

Prodhimet e gramtikës LL(1) janë të përcaktuara si më poshtë:





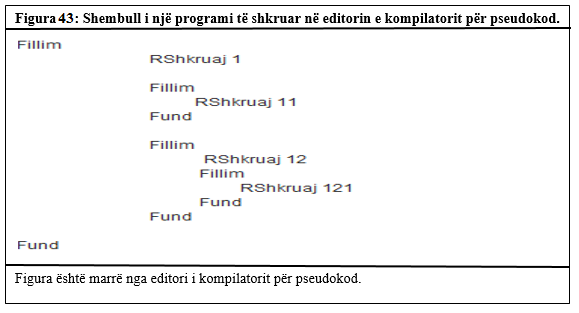


Gramatika e përdorur nga kompilatori për pseudokod ka 22 prodhime.

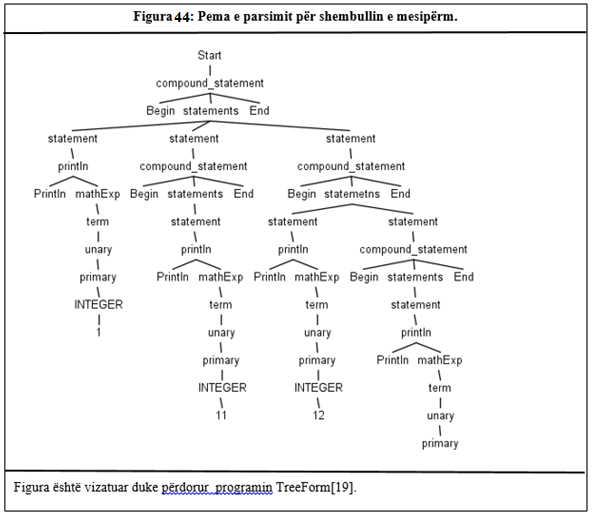
Prodhimi i numër 1 ka simbolin fillestar të gramatikës dhe zëvëndosohet me jo fundorin compound\_statement. Jo fundori compound\_statement mund të zëvëndësohet nga prodhimet e jo fundorit statements të ndërfutur brenda fundorëve Begin dhe End. Jo fundori statements mund të zëvëndësohet nga prodhimi ose prodhimet e një ose më shumë jofundorëve statement. Vetë prodhimi statement mund të zëvëndësohet nga prodhimet compound\_statement, assign, read, print, println, if, while, do\_while, for.

Prodhimi që zëvendëson jo fundorin statement me compound\_statement është përgjegjës që të kemi blloqe instruksionesh që ndërfuten bëenda blloqeve të tjera instruksionesh.

Nuk është e mundur të deklarosh një bllok instruksionesh pa pasur të paktën një instruksion të ndërfutur ndërmjet fundorëve Begin dhe End të bllokut.



Më sipër është një sekuencë tokenash që njihet me sukses si pjesë e gjuhës që përcakton gramatika e psedukodit. Për ta kuptuar më mirë se si prodhimet përcaktojnë nëse sekuenca e mësipërme është pjesë e gjuhës ndërtojmë pemën e parsimit.



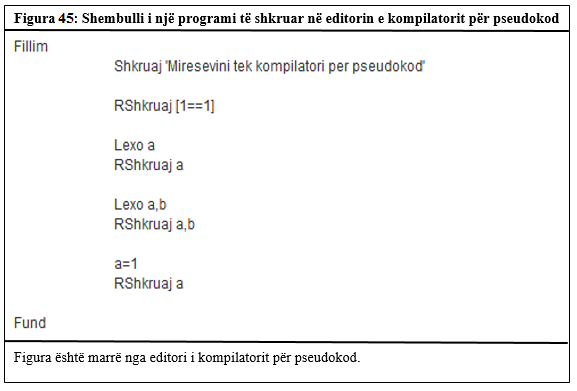
Secili nga jo fundorët assign, read, print, println, if, while, do\_while, for zëvëndësohen nga jo fundorë të tjerë që specifikojnë prodhimet se si duhet organizohet sekuencat e tokenave për të qënë pjesë e gramatikës .

Jo fundori read zëvëndësohet nga fundori Read dhe prej një ose më shumë fundorësh Identifier.

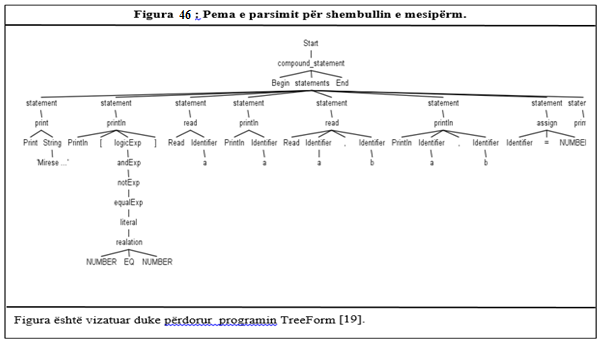
Jo fundori print mund të zëvëndësohet nga fundori Print dhe një ose më shumë argumente. Çdo argument që i kalohet jo fundorit print pas të parit ndahet me presje. Argumentat që mund të kalohen te funksionet print mund të jenë simboli $ dhe Identifikuesi, një jofundor mathexp , një jofundor shprehje logjike e ndërfutur brenda kllapava katrore dhe një String.

Jo fundori println është i ngjashëm me jo fundorit print, i vetmi ndryshim është që fundori nuk është Print por është Println.

Jo fundori assign zëvëndësohet nga fundori Identifier që ndiqet fundori Equal(=) dhe vlera do ti kalohet mund të jetë fundori Identifier , fundori String, jo fundori mathExp ose jo fundori LogicExp i ndërfutur brenda kllapave katrore.



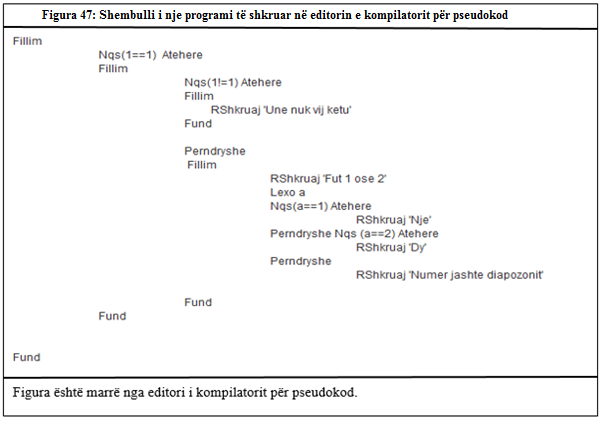
Më sipër kemi një sekuencë me tokena që përdoret për funksionet e leximit ,shkrimit dhe vlerëdhënies dhe për të provuar që janë shkruar në rregull dhe që janë pjesë e gjuhës ndërtojmë pemen e parsimit dhe për të kuptuar se si kompilatori për pseudokod i përkthen nëse janë pjesë e gjuhës apo jo.



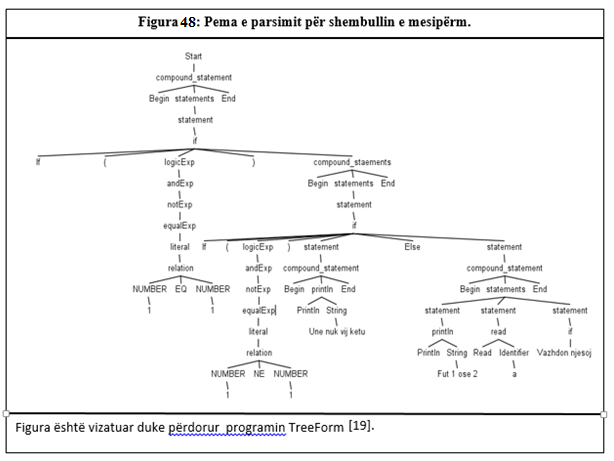
Prodhimet e gramatikës për gjenerimin e shprehjeve aritmetike janë bërë në mënyrë të tillë që të ruajnë precedensën e operatorëve. Operatorët me precedensë më të ulët janë te parët që zëvëndësohen , ndërsa operatorët me precedensë më të lartë janë të fundit që zëvëndësohen.

Prodhimet e gramatikës për gjenerimin e shprehjeve logjike janë bërë përsëri duke ruajtur precedesën e operatorëve logjike të shprehjeve llogjike.Kombinimet e relacioneve që mund të krahasojmë janë numër , varg karakteresh dhe identifikues me përsëri numër, varg karakteresh dhe identifikues. Veprimet e krahasimet që mund të kryhen me operandët e mësipërm janë më madhe se , më vogel se, më madhe e barabarte, m e vogel e barabarte, e barabarte ose ndryshme. Precedensa e operatorëve logjikë si Or, And,Not ruhet duke zëvëndësuar operatorin me prioritet më të lartë të fundit. Ne mund të krahasojmë shprehje logjike me njera tjetrën. Kur si operand kemi shprehje logjike, atëhere numri i veprimeve që mund të kryejme është 2, i barabartë ose e ndryshme.

Jo fundori if është përgjegjës për specifikimin e struktrurave përzgjedhëse për kompilatorin për pseudokod. Ky jo fundor mund të zëvëndësohet vetëm me strukturën përzgjedhëse një drejtimore që mund të jetë një fundor If i ndjekur nga një shprehje logjike brenda kllapave të rrumbullakta dhe jo fundori statement. Mund të jete dy drejtimore që përveç fundor If i ndjekur nga një shprehje logjike brenda kllapave të rrumbullakta dhe jo fundori statement ndiqet edhe nga fundori else dhe një jo fundor statement. Mund të jetë edhe shumë drejtimore ku ndërmjet fundoit If i ndjekur nga një shprehje logjike brenda kllapave rrumbullake dhe jo fundori statement ndiqet edhe nga fundori dhe fundori else dhe një jo fundor statement mund të ketë nga 0 deri shumë jo fundorë else if që ndiqen nga një shprehje logjike e ndërfutur brenda kllapave të rrumbullakta dhe nga jo fundori statement. Jo fundori lejon që të pranohen struktura përzgjedhëse një drejtimore, dy drejtimore dhe shumë drejtimore. Këto mund të jenë të ndërfutura edhe brenda njëra tjetrës.

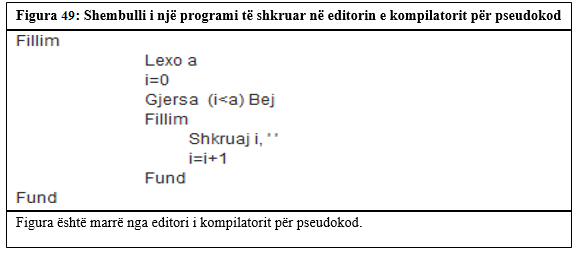


Te shembulli i mësipërm kemi strukturat përzgjedhëse një drejtimore, dy drejtimore dhe shumë drejtimore që janë të ndërfutura brenda njera-tjetrës. Për ta kuptuar si vlerëson kompilatori për psedukod nëse sekuenca e mësipërme është pjesë e gjuhës apo jo, ndërtojmë pemën e parsimit.

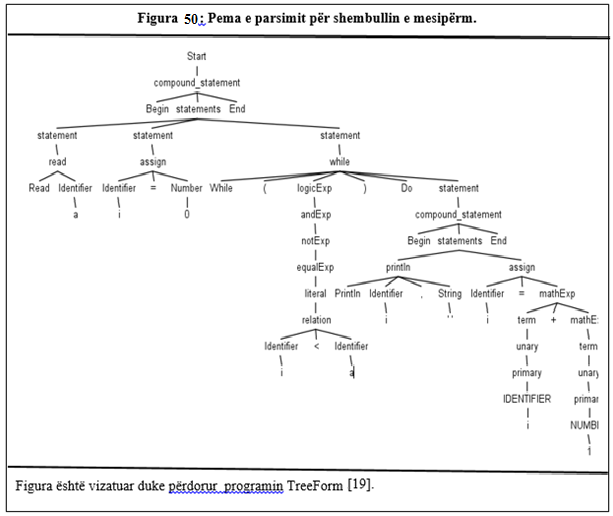


Përveç strukturave përzgjedhëse kemi edhe strukturat përsëritëse që ndryshojnë rrjedhën e kontrollit të programit. Strukturat përsëritëse ekzekutojnë një bllok veprimesh aq herë sa plotësohet kushti. Kompilatori për pseudokod njeh tre tipe strukturash përsëritëse. Këto janë strukturat përsëritëse while, do while, dhe for.

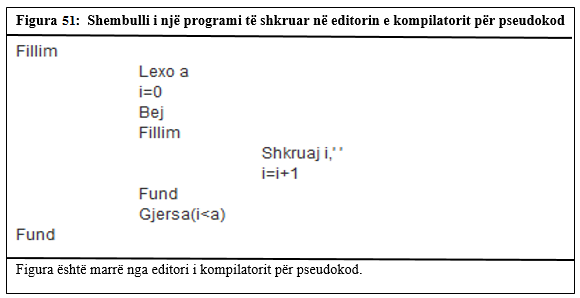
Strukturat përsëritëse while ekzekutojnë një instruksion për aq kohë sa plotësohet kushti. Prodhimet gramatike që specifikojnë ciklin while është jo fundori while. Jo fundori while zëvëndësohet nga fundori While i ndjekur nga jo fundori logicExp i ndërfutur brenda fundorëve kllapa të rrumbullakta dhe më pas ndiqet nga jo fundori statement.



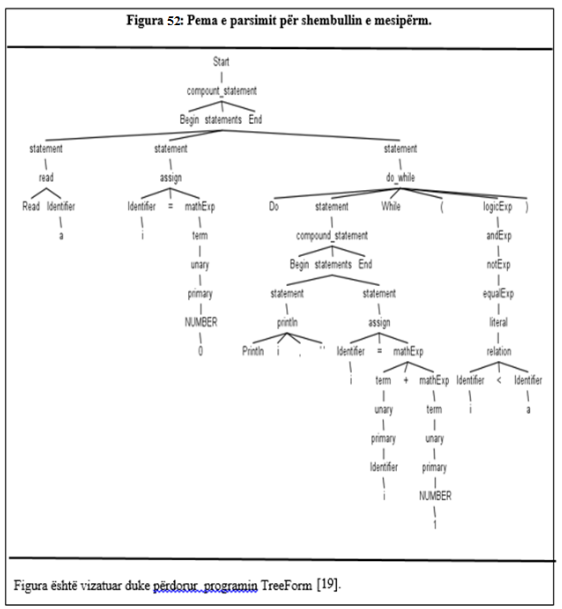
Parseri merr sekuencën e tokenave të mësipërm dhe gjeneron pemën e parsimit nëse kjo sekuencë tokenash i përket gjuhës së përcaktuar nga gramatika e kompilatorit për pseudokod.



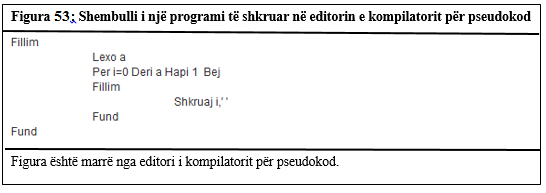
Jo fundori do\_while është përgjegjës për specifikimin e strukturës sintaksore, për strukturat përsëritëse do..whille. Prodhimet gramatikore të do..while janë ato që lejojnë të kemi edhe cikle të ndërfutura brënda njeri-tjetrit dmth brënda ciklit do\_while mund të kemi cikle , struktura përzgjedhëse apo instruksioni të tjera .



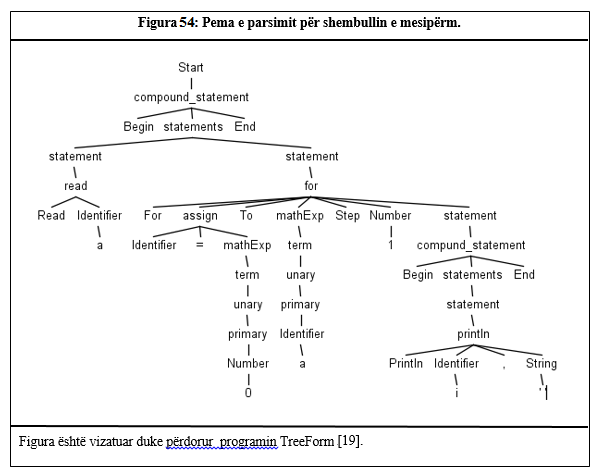
Parseri përsëri sekuencën e mësipërme të tokenave e konverton me pemë parsimi si më poshtë.



Prodhimi for specifikon rregullat sintaksore që duhen të ndjekin tokenat në mënyrë që të pranohen nga parseri i kompilatorit për pseudokod. Prodhimi for lejon që të kemi brenda trupit të for të ndërfuten instruksione , struktura përzgjedhëse ose struktura përsëritëse. Në ciklin for inicialicimi i kushtit dhe përditësimi i ndryshores së kontrollit janë në një komandë e vetme.

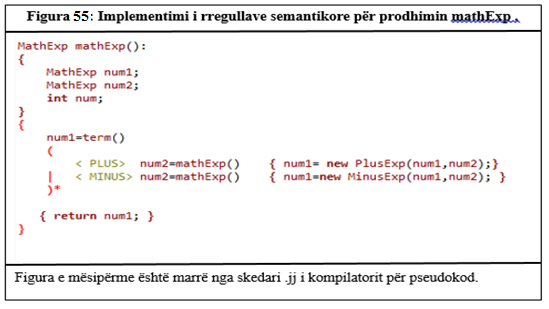


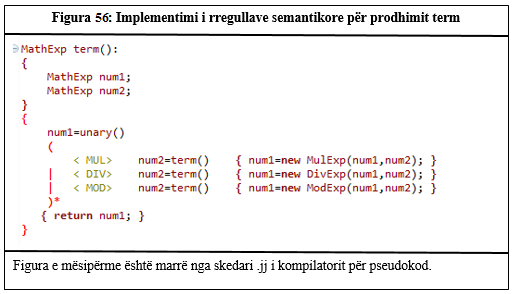
Pema e parsimit që ndërtohet nga parseri është si më poshtë.

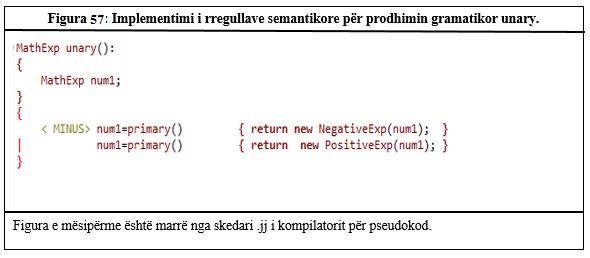


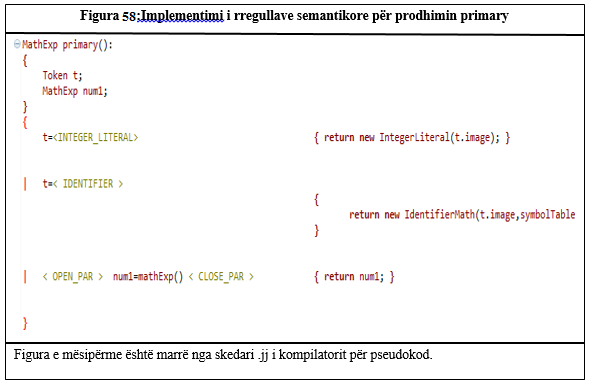
Parseri nga rregullat e gramatikës ka si detyrë vetëm të kontrolloje nëse një sekuencë tokenash ështe pjesë e gjuhës së përcaktuar nga gramatika e kompilatorit për pseudokodin. Për të gjeneruar rezultate nga pemët e parsimit duhet të lidhen me veprimet të prodhime gramtikore. Për një kompilator të thjeshtë mund të ndërfutësh kodin që do të ekzekutohet për një rregull të caktuar gramatikor brenda prodhimit të javacc por kjo është shumë komplekse që parseri dhe semantika e programit të jenë në një vend kur kompilatori është kompleks. Kompilatori për pseudokodin ka përdorur modelin e zhvillimit interpretues për ndarë parserin nga rregullat semantikore.

Paketa Math është përgjegjëse për vlerësimin e shprehjeve aritmetike dhe kthimin e rezultatit pas vlerësimit. Paketa përmban klasa për çdo veprim aritmetikor dhe për operandët e shprehjes. Tek çdo prodhimi gramatikor ne instancojme një klasë që përkon me veprimin që po kryhet.









Paketa Math ka një klasë MathExp që është klasa krysore që trashëgojnë të gjithë klasat e tjera. Kjo klasë përveç se trashëgohet nga të gjitha klasat përdoret edhe si atribut për klasat e tjera. Kjo është një klasë abstrakte ku ka një metode eval që kthen rezultatin pas vleresimit të shprehjes. Rezultati që kthehet është numër i plotë. Klasat PlusExp,MinusExp,MulExp,DivExp dhe ModExp janë klasa për veprimet aritmetike kryesore dhe këto klasa kanë dy objekte të tipit MathExp që të kryejnë veprime aritmetike. Numri atributeve të tipit MathExp për klasat varet nga numri i operandëve që ka veprimi aritmetikor. Klasat NegativeExp dhe PositiveExp janë klasa që kanë vetëm një atribut të tipit Mathexp pasi veprimi i tyre kërkon vetëm një operand.

Më poshtë është diagrama e klasave për vlerësimin e shprehjeve aritmetike. Gjithashtu kemi edhe klasa që nuk kanë atribut të tipit MathExp , këto janë klasa që zakonisht përfaqësojnë operandët e shprehjes dhe detyra e tyre është të kthejnë vlerën e literalit që mund të vijë nga përkthimi i stringut të tokenit në numër ose kërkimin në tabelen e simboleve.

Thelbi i vlerësimit të shprehjeve të rregullta qëndron te rekursion. Gjithmonë rezultati i veprimit më prioritar është i fundit që i thërritet funksioni eval por i pari që ekzekutohet dhe që merr rezultatin.

**Figura 59: Diagrama e klasave për paketën Math**

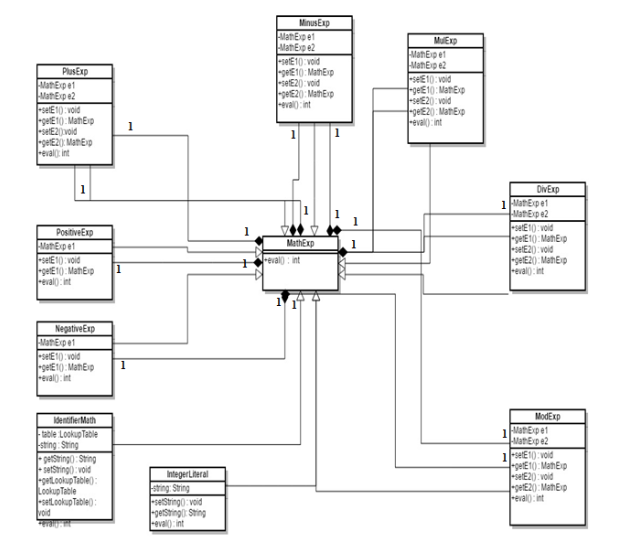
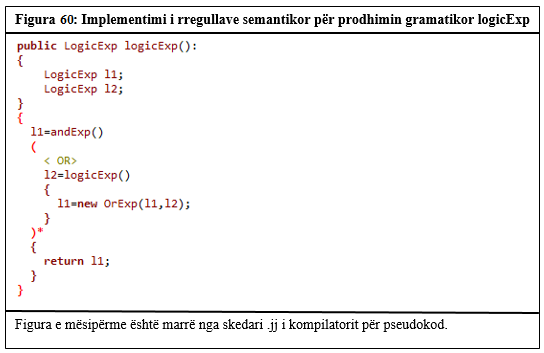
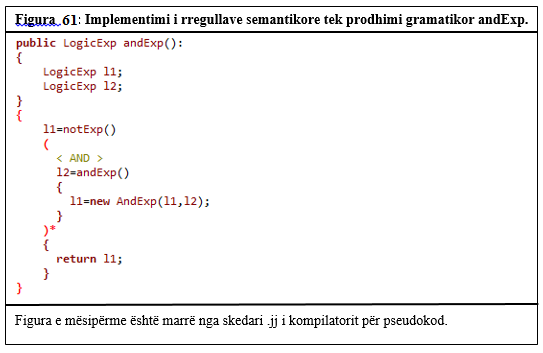
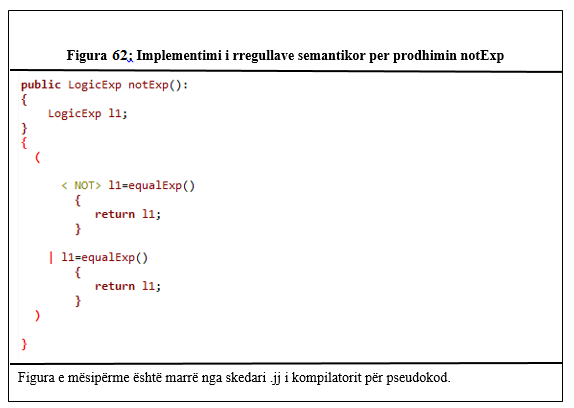


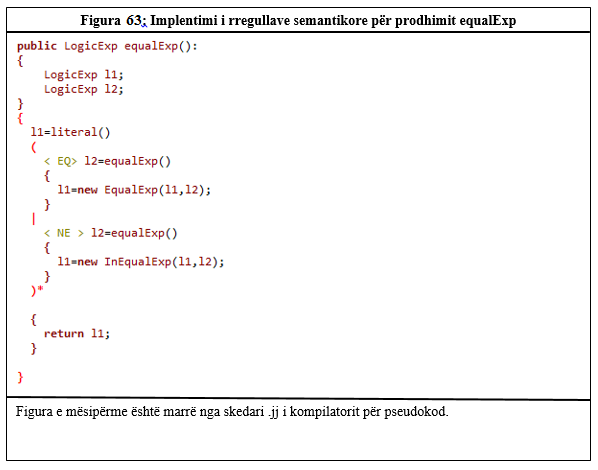
Diagrama e klasave është ndëtuar duke përdour programin violet [20].

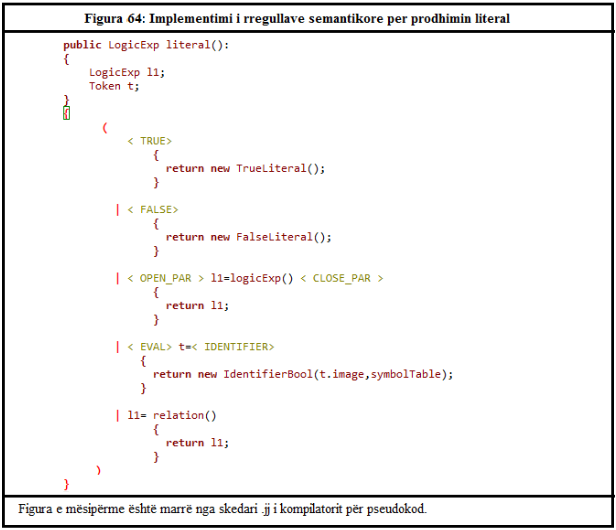
Paketa Logic është përgjegjëse për vlerësimin e shprehjeve logjike dhe kthimin e rezultatit pas vlerësimit. Paketa përmban klasa për çdo veprim logjike dhe për operendët e shprehjes. Te çdo prodhim gramtikor ne instancojmë një klasë që përkon me veprimin që po kryhet.

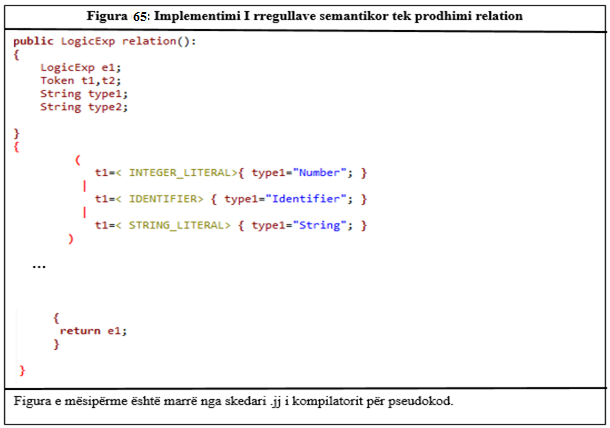










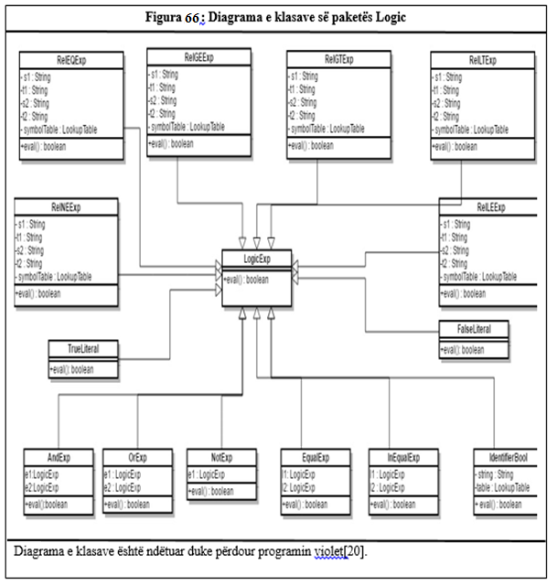


Shprehjet logjike mund të kthejnë vetëm dy vlera : true ose false. Klasa kryesore është klasa abstrakte LogicExp prej të cilave trashëgojnë të gjitha klasat e tjera të paketës Logic dhe ka një metodë eval. Kjo është metodë abstrakte dhe tipi i kthimit është Boolean. Klasat që përfaqësojnë veprime logjike kanë atribute të tipit LogicExp në varësi të numrit të operandëve që kanë.

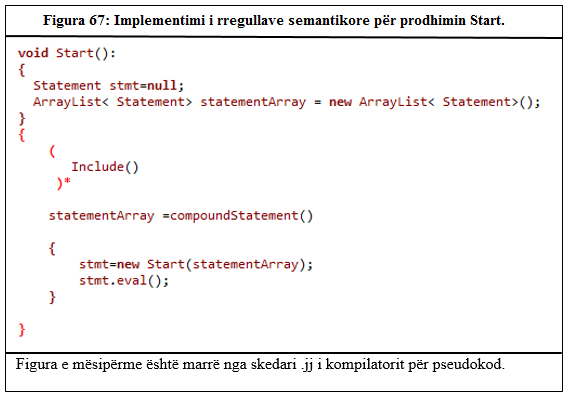
Të gjitha klasat e paketës LogicExp që janë operatorë logjike kanë aq atribute të tipit LogicExp saç kanë edhe operandë.

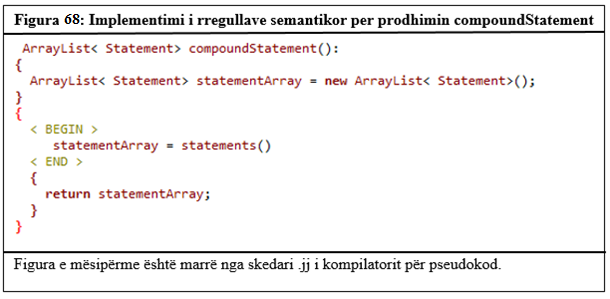
Më poshtë është diagrama e klasave për vlerësimin e shprehjeve logjike. Gjithashtu, kemi edhe klasa që nuk kanë atribut të tipit LogicExp , këto janë klasa që zakonisht përfaqësojnë operandët e shprehjes dhe detyra e tyre është të kthejnë vlerën e literalit.

Thelbi i vlerësimit të shprehjeve logjike qëndron në rekursion. Gjithmonë rezultati i veprimet me prioritet më të lartë është i fundit që i thërritet funksioni eval por i pari që ekzekutohet dhe që merr rezultatin.



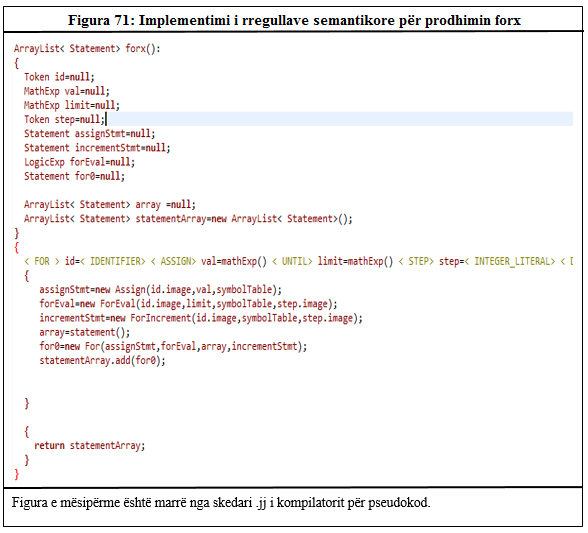
Paketa Statement ka klasat që merren me interpretimet e instruksioneve të leximit , afishimit dhe vleredhenies. Gjithashtu paketa Statement është përgjegjëse për implementimin strukturave përzgjëdhëse dhe strukturave përsëritëse.

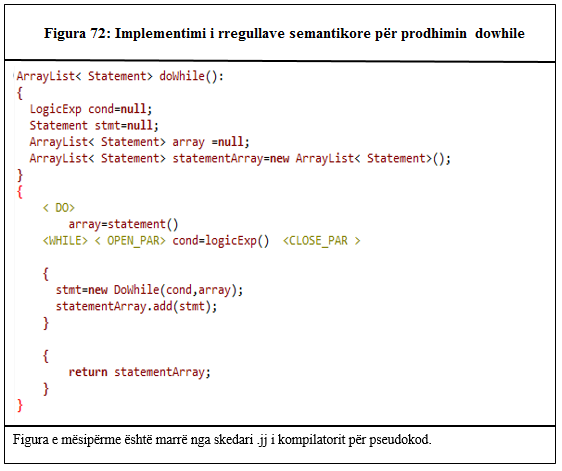


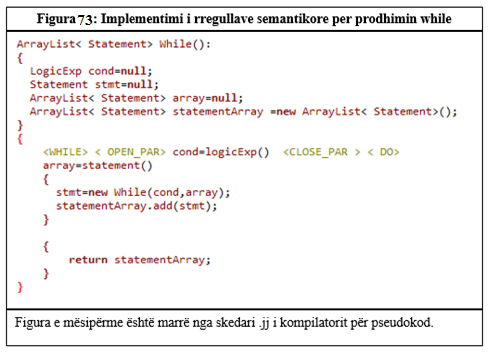


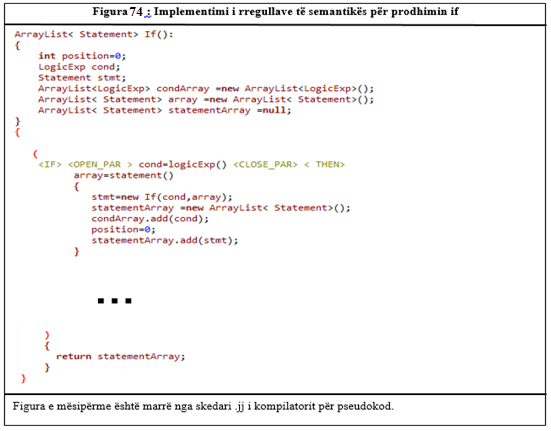


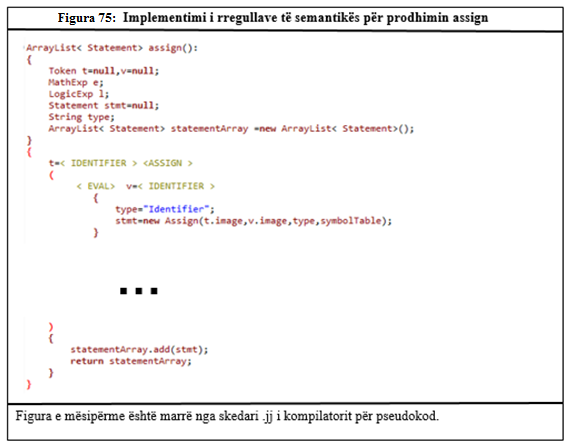


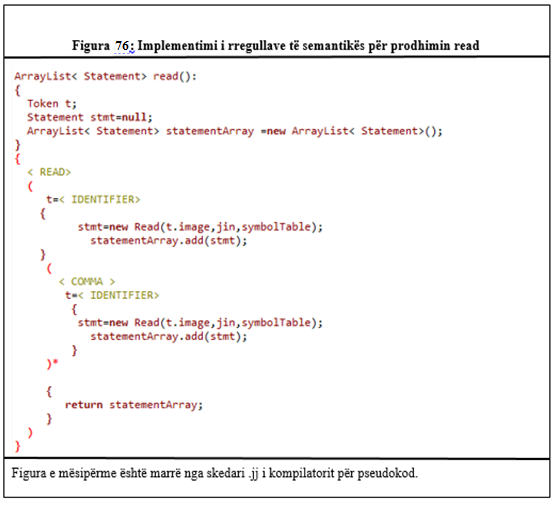


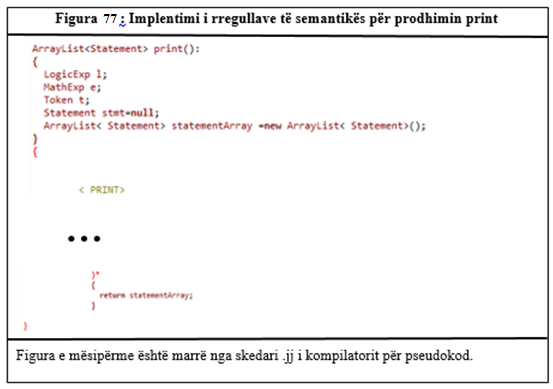


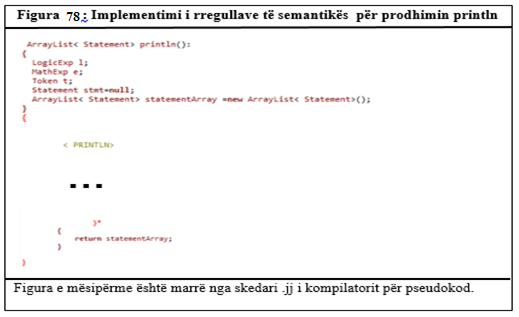












Klasa kryesore e paketës statement është klasa Statement. Kjo është klasë abstrakte dhe nga kjo klasë trashëgojnë të gjithë klasat e tjera. Për secilin nga instruksionet kemi një klasë të veçante që ka të implementuar metodën eval të klasës abstrakte Statement. Kjo metodë eval ekzekuton isntruksionin për afishimin, leximin ose vleredhënien. Të gjithë instruksionet vendosen në një listë dhe më pas ekzekutohen. Në këtë paketë janë implementuar edhe klasat për implementimin e strukturave përzgjedhëse dhe përsëritëse. Klasat që implementojnë strukturat përzgjedhëse caktojnë cilin instruksion do të shtojnë te lista që përmban instruksionet që do të ezkekutohen. Ndërsa klasat e strukturave përsëritëse shtojnë instruksionet te lista që permban instruksionet që do të ekzekutohen aq herë saç plotësohet kushti.

Mëposhte është diagrama UML e klsave të paketës Statement.

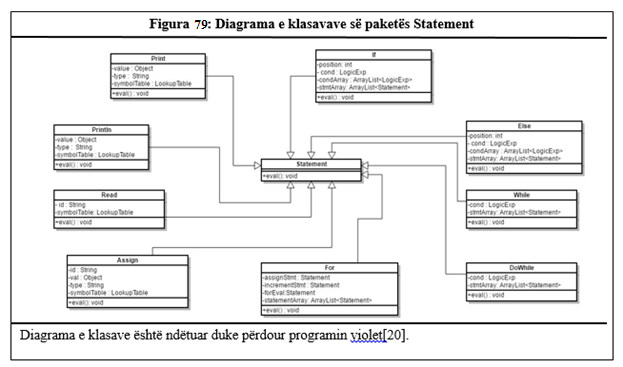
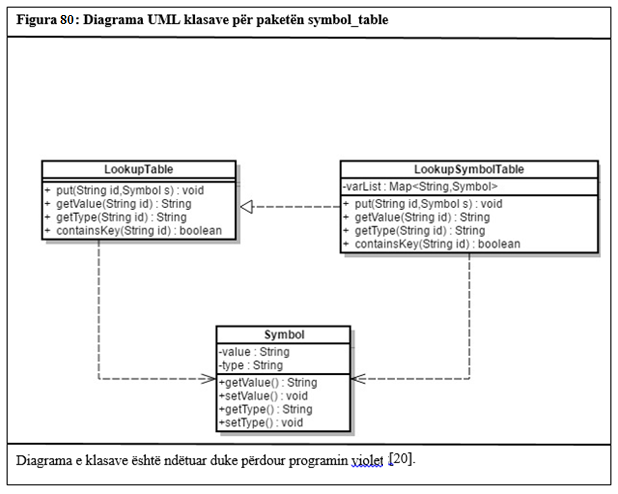
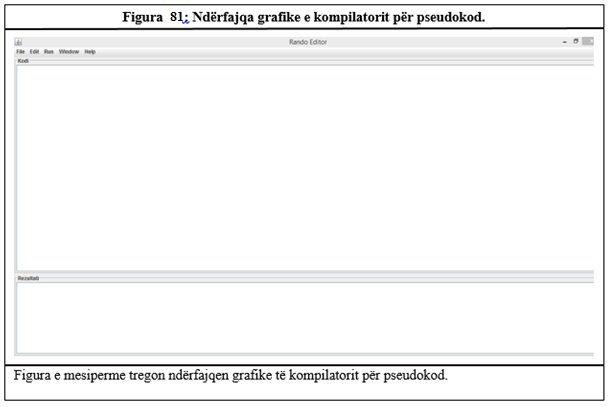


Tabela e simboleve ruan vlerat për çdo ndryshore që është inicializuar , n.q.s një ndryshore përdoret për here të parë atëhere ajo shtohet në tabelën e simboleve me vlerën default të saj , që për numrat është 0 , për vargjet e karaktereve është bosh dhe për vlerat logjike është false. Paketa symbol\_table është përgjegjëse për implementimin e strukturës së të dhënave për tabelën e simboleve që i duhet kompilatorit për të marre , ruajtur , aksesuar vlerat e ndryshoreve.Tabela e simboleve ka një klasë që quhet Symbol që është përgjegjëse për rekordet që do të ruhen në këtë tabelë . Ajo ka atributet ,vlerë dhe tipin për çdo ndryshore që deklarohet. Një ndryshore quhet e deklaruar në momentin që ajo fillon të përdoret në program. Strukturat e të dhënave përdorur për implementimin e tyre është një HashMap , ku çdo çelës shoqërohet me një vlerë të caktuar. Te tabela jonë e simboleve, çelësi është identifikuesi i ndryshores ndërsa vlera është objekti i tipit symbol që përmban tipin dhe vlerën për ndryshoren korresponduese.

Më poshtë është diagrama uml e klasave të paketës symbol\_table.

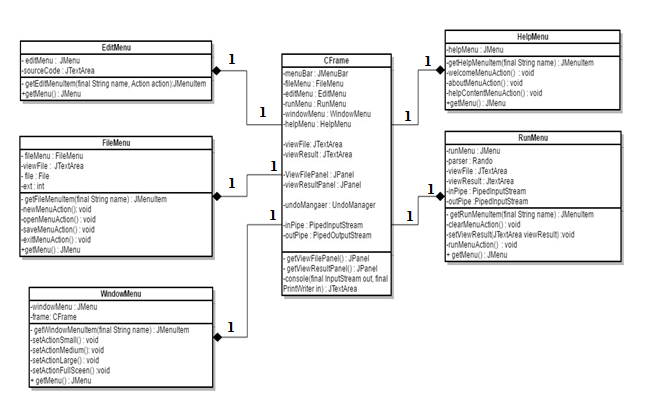


Kompilatori për pseudokod ka një ndërfaqe grafike që lehtëson punën e programuesit për të shkruar kodin. Ajo ka një editor ku shkruan kodin, dhe rezultati i shfaqet poshtë editorit. Ndërfaqja grafike i mundëson të ruajë skedarin me programin, të hapë skedar të rinj. Gjithashtu i lejon të bëjë undo dhe redo. Ka një menu help ku mund të gjejë informacione mbi përdorimin e kompilatorit për pseudokod. Dhe gjithashtu i jep mundësi të kontrollojë madhësine e nderfaqes grafike që do të përdoret për të shkruar kodin. Për të ekzekutuar kodin është menuja run me dy funksionet .Run që ekzekuton kodin dhe clear që pastron fushën e tekstit ku shfaqet rezultati.



Paketa gui mban të gjitha klasat përgjegjese për ndërfaqen grafike. Klasa më e rendësishme prej tyre është klasa CFrame ku mbi të vendosen të gjithe elementet e tjerë të grafikës si: menutë, editorët dhe framet e tjerë. Më poshtë kemi paraqitjen në UML të klasave të paketës gui.

**Figura 82: Diagrama e klasave së paketës gui**

Diagrama e klasave është ndëtuar duke përdour programin violet[20].

## 4.3. Dokumenti i vendimeve kyçe të marra gjatë zhvillimit

Vendimet më të rëndësishme për këtë temë janë:

* Gjuha mbi të cilën do të ndërtohej kompilatori do të ishte java dhe si mjet që do të përdorej për gjenerimin e analizuesit leksikor dhe parserit do të ishte javacc.
* Tipi i parserit do të ishte parser të tipit top down, ngaqë javacc gjeneron parser të tipit top down dhe këto lloj parserash përdoren mbi gramatika të llojit LL(k).
* Nje ndër vendimet kyçe që është marrë në ndërtimin e kompilatorit për pseudokod ka qënë ndarja e analizes sintaksore nga analiza semantikore. Duke përdorur paketat e pemës abastrakte të sintaksës, pjesa e kodit që gjeneron rezulatin është ndarë nga parseri që kontrollon nëse një sekuencë tokenash është pjesë e gjuhës apo jo.
* Një tjetër vendim kyç ishte edhe implementimi i paketës për tabelën e simboleve. Në fillim tabela simboleve u implementua duke përdorur një hashMap nga framework i strukturave të të dhënave të javes, por më pas u mor vendimi që implementimi i tabelës se simboleve do të bëhej paketë më vete.
* Shprehjet aritmetike kthejnë numra të plote ndërsa shprehjet logjike kthejnë vlera logjike, prandaj nuk mund të vazhdonte gramatika në mënyre hiearkike. Për të bere lidhjen ndërmjet tyre u perdorur një produksion relation. Ky produksion bashkon tre tipet e te dhënave numra, stringje dhe vlera logjike. N.q.s tipet ndryshojnë, nuk hedh gabim por kthen false direkt, kur kanë tipe të njëjta atëhere vazhdon bën një kontroll nëse vlerat janë të njëjta.
* Problemi në printimin dhe vlerëdhenien është perplasja që has shprehjet aritmetike me shprehjet logjike. Për t’i dalluar ato , është vendosur që shprehjet logjike te jenë të ndërfutura brenda kllapave katrore [ shprehje logjike].

# 5. Rezultate

## 5.1. Hyrje

Rezultati i kësaj pune është një kompilator që njeh një gjuhë të re programimi.Si çdo gjuhë programimi kjo ka sintaksën saj për shkrimin e instruksioneve bazë ,strukturave përzgjedhëse dhe përsëritëse. Në këtë kapitull do të trajtojmë të gjithë rregullat sintaksore që duhen ndjekur për të shkruar saktë programe në gjuhën e kompilatorit për pseudokod. Përveç se do të shpjegohet çdo funksionalitet i programimit me detaje, gjithashtu do të kemi nga një shembull të shkruar në gjuhën e kompilatorit për pseudokod së bashku me rezultatin e gjeneruar prej programit. Në këtë kapitull do të përshkruhet me detaje rezultati i punes duke shpjeguar të gjitha opsionet që ofron gjuha e re e programimit.

## 5.2. Instruksionet bazë

Shkruaj është funksioni që printon informacionin që shfaqet në hapësirën e rezultatit. Mund të marrrë disa parametra të tipeve të ndryshme që ndahen me presje.

RShkruaj është funksioni që printon informacionin që shfaqet në hapësirën e rezultatit, por në dallim nga Shkruaj , kursori lëviz në rresht te ri.

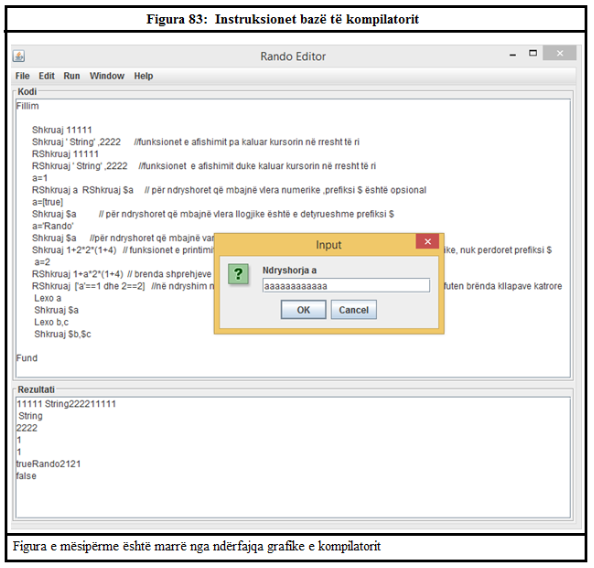
Ndryshoret përdoren për të ruajtur informacion, në këtë gjuhë nuk kërkohet të specifikohet tipi i ndryshores.Një ndryshore mund të marrë vlera të tipeve të ndryshme. Për të printuar ndryshore që kanë vlera numerike nuk është e nevojshme të kemi simbolin $, për ndryshoret që janë stringje ose vlera logjike duhet patjetër të shoqërohen me prefiksin $.

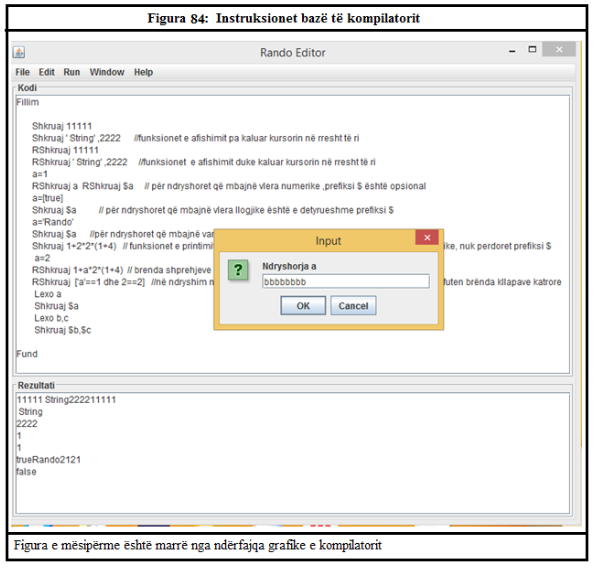
Printimi i shprehjeve aritemtike. Precedensa e vlerësimit të shprehjeve aritmetike është si më poshtë\*,/,% dhe +,- . Shprehja që ndodhet brenda kllapave do të vlerësohet e para dhe ajo ka prioritet më të lartë.

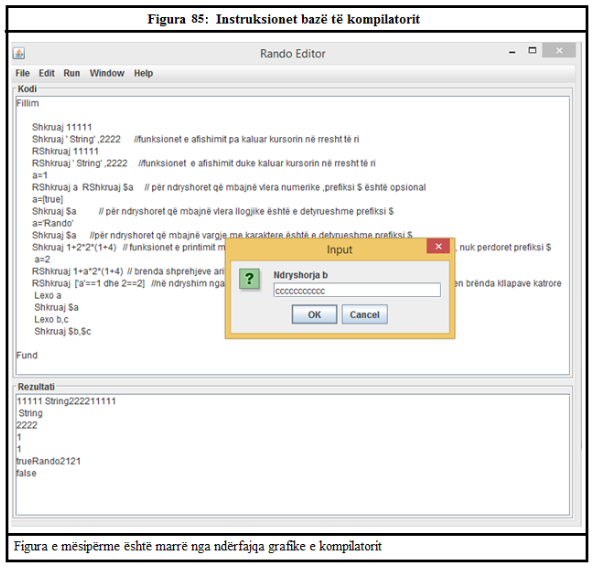
Printimi i shprehjeve aritmetike që kanë edhe ndryshore. Një ndryshore që ka vlerë numerike mund të jetë pjesë e shprehjes aritmetike si çdo shifër.

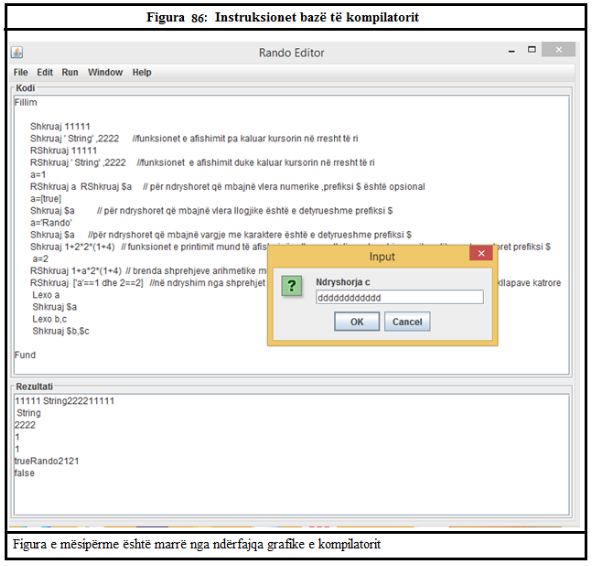
Printimi i shprehjeve logjike bëhet duke vendosur shprehjen logjike brenda kllapave katrore.Në shprehjet logjike mund të krahasohen ndryshore me vlera të ndryshme , por nëse kanë tipe të ndryshme do të kthehet false.Për të caktuar diferencë ndërmjet ndryshoreve që kanë vlera logjike (boolean ) dhe për ndryshore të tipit të tjera përdoret simboli $. Ndryshoret me vlera boolean kanë simbolin $ si prefiks për t'i dalluar nga ndryshoret që nuk mbajnë vlera logjike.Për të lidhur dy shpreje logjike bashkë përdorim lidhëse dhe , ose. Për të marrë të kundërtën e vlerës përdorim operatorin e mohimit !.

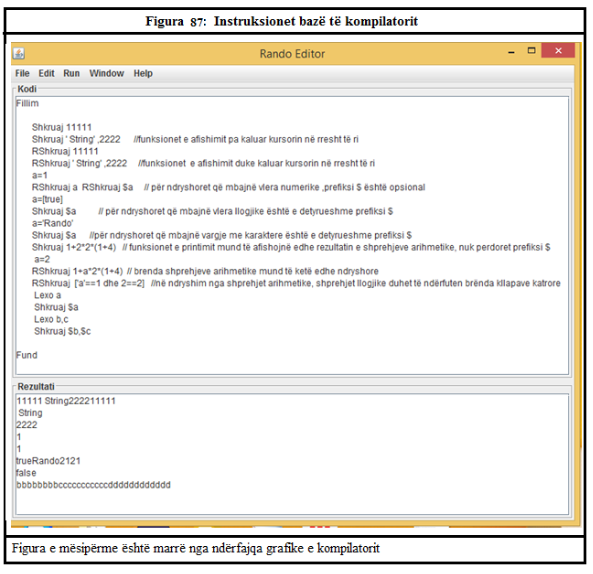
Për të lexuar të dhëna nga përdoreusi përdoret funksioni Lexo dhe caktohen ndryshoret që do të lexohen. Nëse kemi më shumë se një ndryshore që duam të lexojmë, atëhere ndryshoret i ndajmë nga njëra -tjetra me presje.





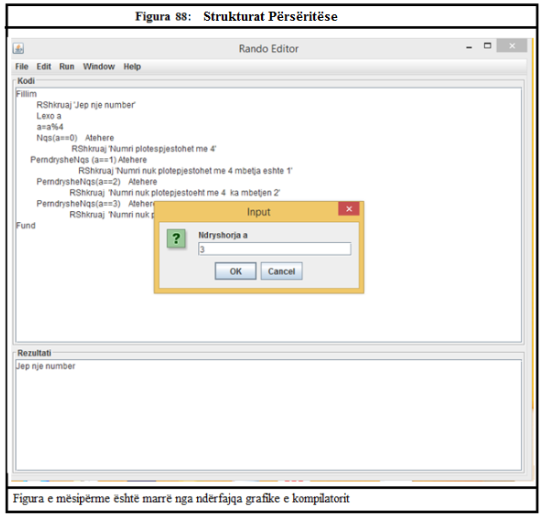


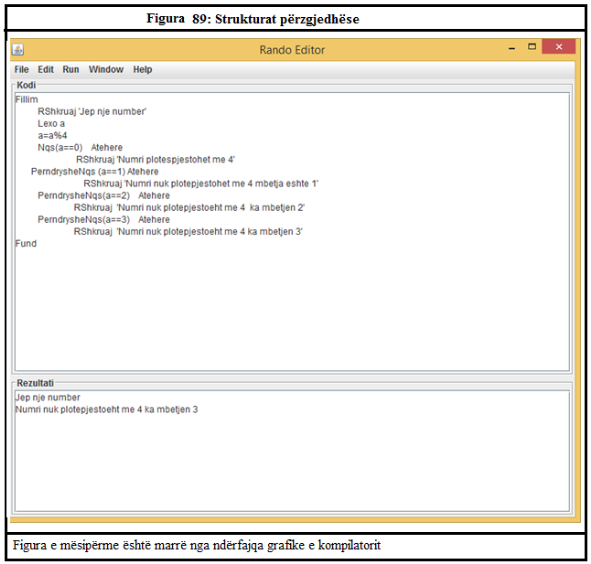




## 5.3. Strukturat Përzgjedhëse

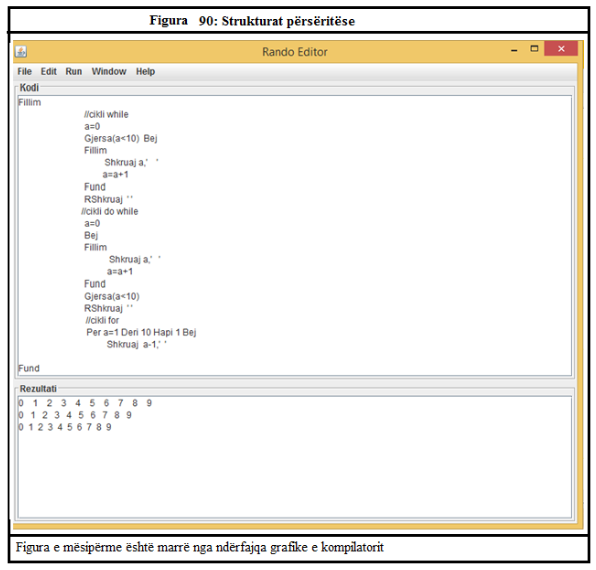
Strukturat përzgjedhëse Nqs , Përndryshe Nqs përdoren për të përcaktuar ekzekutimin e një instruksioni mbi bazën e kushtit që është vendosur në strukturat përzgjedhëse.





## 5.4. Strukturat Përsëritëse

Strukturat përsëritëse që ofron ky kompilator janë Gjersa, Bëj Gjersa, Për Deri . Strukturat përsëritëse përsërisin ekzekutimin e një instruksioni aq herë saç plotësohet kushti.



# 6. Konkluzione

Kompilatori për pseudokod ka të implementuar:

* Instruksionin e leximin nga përdoruesi të të dhënave, përdoruesi mund të japë te dhëna duke përdorur disa herë instruksionin e leximit, ose duke përdorur një here dhe duke i ndare me presje identifikuesit që do të ruajnë të dhëna në instruksionet e leximit.
* Instruksionet e afishimit. Kemi dy tipe instruksionesh afishimi , instruksioni që kalon kursorin në rresht të ri dhe instruksioni i afishimit që nuk e kalon kursorin në rresht të ri. Ne mund të afishojmë vlerën e një identifikuesi, varg karakteresh , shprehje aritmetike dhe shprehje logjike.
* Instruksioni i vlerëdhënies i cakton një vlerë një ndryshoreje. Ndryshorja të cilës do i caktohet vlerë në instruksionin e vlerëdhenies është gjithëmonë në të majtë, ndërsa vlera që do t'i caktohet është në të djathtë. Vlera që mund të marrë një ndryshore mund të jetë varg karakteresh, rezulati i një shprehje aritmetike, rezultati i një shprehje logjike ose dhe vetë vlera e një identifikuesi tjetër. Ndryshoret nuk kanë nevojë të deklarohen, ato janë të deklaruara në momentin që përdoren. Tipi i ndryshoreve caktohet në mënyre implicite , ajo përcaktohet nga vlera që i kalohet ndryshores.
* Strukturat përzgjedhëse që modifikojnë rrjedhën e ekzekutimit të programit në varësi të kushtit që i kalohet. Një instruksion ekzekutohet nëse plotësohet kushti i vendosur në strukturën përzgjedhese. Strukturat përzgjedhëse që implementon kompilatori për pseudokod janë kushtet if..else if … else .
* Strukturat përsëritëse që gjithashtu modifikojnë rrjedhën e ekzekutimit të programit , duke e ekzekutuar një instruksion aq herë sa plotësohet kushti. Strukturat përsëritëse që janë të implementura në kompilatorin për pseudokod janë while , do-while dhe for.
* Në kompilatorin për pseudokod mund të krahasosh vlerat e ndryshoreve pavaresisht tipit , vargjeve me karaktereve, numrave me vlerat e ndryshores pavarësisht tipit, vargjeve me karaktere dhe numrave. Numri i kombinimeve është shumë i madh , po rregullat e vendosura nga ana e semantikës janë që momentin që krahasojmë ndryshoret, varg karakteresh ose numer nëse nuk eshte i njejti me operandin në të djathtë atëhere automatikisht vlera e shprehjes logjike do të ishte false.

Për të realizur kompilatorin për pseudokod është përdorur mjeti JavaCC. Me anë të javacc është gjeneruar analizuesi leksikor duke përcaktuar shprehje të rregullta për çdo token dhe parseri duke specifikuar prodhimet gramatikore. JavaCC gjeneron një parser të tipit top-down dhe parserat e tipit top-down kanë specifike që gramatika që ato implementojnë duhet të jetë patjeter e klasës LL(k). Që do të thotë që duhet të eliminojë faktorizimin në të majte dhe rekursionin në te majtë. Kjo ka sjelle problem sidomos kur duhet të vlerësojë prodhime gramatikore që fillonin me të njejtin token dhe parseri gjithmone e bënte zëvëndësimin me prodhimin e parë. Për ta eliminuar këte u bene disa modifikime për të dalluar prodhimet gramatikore nga njëra tjetra.

Për të implementuar rregullat semantike është përdorur modeli i interpretuesit , i cili ndan parserin nga rregullat semeantike që prodhojnë rezultatin nëse programi është shkruar në rregull. Megjithëse kompilatori për pseudokod nuk eshte një kompilator me funskionalitete komplekse siç janë kompilatoret e gjuhes C ose C++ , përseri ishte shumë e komplikuar që rregullat semantikore të implementoheshin së bashkur me parserin. Dhe përveç faktit që ishte shumë e veshtirë , nuk ishte as menyra më e mire per te zgjidhur problemin pasi gjetja e bugeve dhe identifikimi i tyre nese ishin gabime gjate analizes leksikore apo analizes semantikore do te ishte shume e veshtire.

Në kompilatorin për pseudokod tabela e simboleve është përdorur për të ruajtur vlerat e ndryshoreve të deklaruara në program. Duke qenë se kompilatori për pseudokod nuk i ka të implementuar funksionet , skopi ndryshoreve vazhdon te qendrojë i alokuar deri në fund të përfundimit të ekzekutimit te programit. Kjo nuk është mënyrë e mirë pasi i bllokon burim për një kohë të gjatë, kur nuk është më e nevojshme.

Për t’u siguruar mbi saktësinë e kompilimit të programeve nga kompilatori për pseudokod është perdorur JUnit Test , të cilat për cdo funksionalitet të kompilatorit kanë nje bashkesi klasash që testojnë rastet e mundshme të programeve që japin rezultatin që është parashikuar.

Nderfaqja grafike e kompilatorit për pseudokod është bërë duke përdorur frameworkun grafik të paketës swing të gjuhës së programimit java. U zgjodh ky framework për shkak të lehtësise që ofron në ndërtimin e grafikave të thjeshta .

# 7. Punë që mund të zhvillohen në të ardhmen

Në të ardhmen kompilatori për pseudokod mund të zgjerohet duke implementuar:

* Krijimi i funksioneve që mund të thirren te funksioni krzesor.
* Përfshirja e librarive të tjera me funksione.
* Tabelat nje përmasore, dy përmasore dhe shume përmasore.
* Shenjuesit, listat një drejtimore , dy drejtimore, rrethore.
* Stivat.
* Radhët.
* Pemët.

Gjithashtu, tabela e simboleve do të ndryshohej që të kishte mundësi të ruante skopin e ndryshoreve dhe në momentin që nje ndryshore del jashtë skopit, atëhere ajo liron vendin e zënë ne memorie.

Gramatika e gjuhës do të thjeshtohej pak më shumë në menyrë që të mos jetë nevoja të nderfuten shprehjet logjike brenda kllapave katrore dhe instruksionet e afishimit dhe vlerëdhënies të mos kenë nevojë për simbolin e vlerësuesit për të afishuar vlerat e identifikuesve.

Ndërfaqja grafike mund të përmirësohet duke përdorur frameworkun më te ri grafik te javes FX që sjell një përmirësim të grafikës në krahasim me swing.

Kompilatori për pseudokod mund të zgjerohej më tej që të mos ishte më vetëm për pseudokod por edhe të kompilonte bllokskema, në këtë rast input i tokenave nuk do ishin karaktere por do të ishin karaktere të shoqëruara me figura gjeometrike.

Dhe si urë lidhëse mund të ndërtohej nje opsion që të konvertonte pseudokodin në bllokskemë dhe bllokskemën në pseudokod.

# 8. Referenca ose Bibliografia

1. Compilers – Principles, Techniques, and Tools; 2nd ed., A. Aho, M.S. Lam, R. Sethi, J.D. Ullman, Addison-Wesley, 2007.pdf
2. http://www.tutorialspoint.com/compiler\_design/compiler\_design\_overview.htm
3. http://techwelkin.com/compiler-vs-interpreter
4. http://www.oracle.com/technetwork/java/javase/downloads/jre7-downloads-1880261.html
5. http://www.tutorialspoint.com/compiler\_design/compiler\_design\_runtime\_environment.htm
6. http://www.tutorialspoint.com/compiler\_design/compiler\_design\_symbol\_table.htm
7. http://www.tutorialspoint.com/compiler\_design/compiler\_design\_code\_optimization.htm
8. Libri compiler construction using java,javaccc,and yacc nga autori Reis
9. http://www.engr.mun.ca/~theo/JavaCC-Tutorial/javacc-tutorial.pdf
10. http://www.ibm.com/developerworks/data/library/techarticle/dm-0401brereton/
11. Analiza Leksikore + Shprehjet e rregullta , autor Klesti Hoxha, pedagog në Fakultetin e Shkencave të Natyrës.
12. Libri Modern Compile Implemtation In Java nga autorët Appel dhe PalsBerg
13. http://www.tutorialspoint.com/compiler\_design/compiler\_design\_intermediate\_code\_generations.htm
14. <http://www.tutorialspoint.com/compiler_design/compiler_design_code_generation.htm>
15. https://en.wikipedia.org/wiki/JavaCC
16. https://en.wikipedia.org/wiki/Java\_(programming\_language)
17. https://en.wikipedia.org/wiki/Eclipse\_(software)
18. https://marketplace.eclipse.org/content/javacc-eclipse-plug
19. https://sourceforge.net/projects/treeform/
20. http://alexdp.free.fr/violetumleditor/page.php