UNIVERSITETI "UKSHIN HOTI"
FAKULTETI I SHKENCAVE KOMPJUTERIKE
DIZAJN SOFTUERI


# PUNIM DIPLOME

**Alixhan Basha**

Prizren, 2020

UNIVERSITETI "UKSHIN HOTI"
FAKULTETI I SHKENCAVE KOMPJUTERIKE
DIZAJN SOFTUERI


**TITULLI I TEMES**

# Krijimi i një gjuhe Programuese

*Mentori*
Prof. Ass. Dr. Arsim SUSURI

*Autori*
Alixhan BASHA

Prizren, 2020

*Falenderime*

Falenderoj profesorin dhe mentorin tim Prof. Arsim Susuri, për përkrahjen dhe kontributin e dhënë në të gjitha fazat e realizimit të këtij punimi.

Falenderoj nga shpirti familjen time, që në kohrat më të vështira kanë qenë pranë meje dhe më kanë mbështetur në cdo situatë. Fakti që sot jam këtu, është falë besimit, inspirimit dhe motivimit që më keni dhënë ju.

Së fundmi falenderoj shoket dhe shoqet që më kanë motivuar dhe ndihmuar gjatë viteve që ishim së bashku. Shpresoj që lidhjet që krijuam do të zgjasin deri në fund.

# Përmbajta

*Abstrakti*

Ky punim diplome përmban sqarime lidhur me krijimin e një gjuhe programuese. Nuk do të përdoren programet si ANTLR, YACC ose Bison, të cilat janë programe që gjenerojnë parsera për gjuhë programuese, gjithcka do të implementohet nga fillimi. Në këtë punim do të flitet për teknologjitë që i enkapsulon gjuhët programuese. Do diskutohen temat si: historia e programimit dhe gjuhëve programuese, cka është në esencë, si funksionon, si vjen deri tek krijimi i një programi, dhe shumë tema të ndryshme. Do të mundohem ta ndriqoj këtë temë në menyrë sa më të thjeshtë dhe të detajuar. Së bashku me spjegimin e teorisë, do të implementohet edhe verzioni ynë i një gjuhë programuese të ashtuquajtur "ALP" ( shkurtesë për "Albanian Programming Language" ).

ALP do të jetë një gjuhë programuese e interpretuar, dinamike, e orientuar në objekte dhe e implementuar në Java. Do ta këtë sintaksën në gjuhën tanë amëtare, pra në gjuhën Shqipe. Do të jetë e lehtë dhe e kapshme për programerët fillestarë të grupmoshave të ndryshme. Do të jetë krejtësishtë e pamvarur dhe "Touring Complete", që do të thotë se është në gjendje të manipulojë të dhënat. Arsyeja se pse është zgjedhur Java si teknologji për implementim është JVM( Java Virtual Machine ) dhe aftesitë e saj, poashtu Java është një gjuhë programuese e cila suportohet nga një numër i madhë i pajisjeve kompjuterike. Përndryshe ky projekt do të ishte shumë i sofistikuar, pasi do nevoitej implementimi i shumë funksioneve të JVM nga fillimi, dhe mbështetja e një numri të madhë të makinave kompjuterike.

*Abstract*

This diploma paper contains explanations about creating a programming language. We will not be using parser generators like ANTLR, Yacc or Bison and everything will be made from scratch. In this paper we will be exploring the technologies that encapsulate programming languages. We will discuss topics like: history of programming and programming languages, what is it in essence, how does it work, how does the code we write gets understood by the computer, and many more. Together with the theory, we will also implement our own version of a programming language so called ALP (Albanian Programming Language).

The ALP programming language will be an interpreted, dynamic language implemented in Java. It will have a syntax based on the Albanian language. It shall be easy to use for programmers of any age and it is going to be independent and Touring Complete, which means that it can handle manipulation of data. The reason that we use Java as our implementation language is the JVM( Java Virtual Machine ) and its capabilities, also Java is supported by a wide range of computer devices. Otherwise this project would have been very sophisticated, since we would need to implement most of the JVM's functionality and we would have to add support for multiple systems.

# Kapitulli 1

# Hyrja

Historia e programmimit fillon që nga kohët e makinave kompjuterike të para dhe është një nga teknologjitë më të zhvilluara. Në fillim programimi ishte vetëm një menyrë e specializuar e reprezentimit të formulave matematikore dhe shkencore, mirëpo me investimin në teknologjinë e kompajllereve dhe përmisimin e hardware-it u krijuan gjuhët programuese të nivelit të lartë që teorikisht mund të përdoreshin për zgjidhjen e cdo problemi

## 1.1 Historia e Programimit

Edhe para se të imagjinohej krijimi i një makine që sot nihet si kompjuter, matematicientet mundoheshin ti reprezentonin problemet e tyre në mënyren që sot nihet si programatike. [1]Në fakt, matematicientja Ada Lovelance ishte personi i parë që krijoji dhe publikoi ”programin” e parë kompjuterik në vitet 1842-1849 . Ishte një program që mund të kalkulonte numrat e Bernoullit, dhe sot nihet si “Analytical Engine”. Kuptohet se kjo ishte vetëm një provë e konceptit e cila e hapi derën për idenë që mund të krijoheshin makina që do të mund të përdoreshin për kalkulime shkencore, dhe ndoshta edhe më shumë. Si babau i shkencave kompjuterike dhe kompjutimit është matematicienti Alan Turing, i cili ka kontribuar jashtë mase në këtë degë, me publikimet e tij të shumta. Mirëpo njera nga ato publikime do të ishte më e rrëndesishmja dhe kjo nihet si ”Turing Machine”. Një makinë abstrakte matematikore për kompjutim që sot është baza e të gjitha gjuheve programuese. Nëse një gjuhë programuese mund të simuloj një Turing Machine, atëherë teorikisht mund të reprezentoj secilin funksionalitet që mund ta kompletoj një makinë kompjuterike.

Kompjuterët e parë ishin të mëdhenjë, jo praktik dhe poashtu ishin shumë të veshtirë për t’u programuar. Secila makinë kompjuterike kishte arkitekturë të ndryshme, dhe kjo do të thotë se për secilin kompjuter nevoitej një programer që e dinte funksionin e makines, dhe programet duhej të shkruheshin për secilën makinë ndamas. Mënyra e programimit në fillim bëhej drejtëpërdrejtë me numra binarë, programeret kishin qasje në cdo pjesë të hardware-it dhe secili bit ishte i rëndësishëm pasi memorja kishte kosto të lartë. Programeret shpejtë e kuptuan se kjo menyrë e zgjedhjes së problemit nuk ishte aspak efikase dhe filluan të mendonin për metoda më të mira. Lind koha e Assembler-ave, gjuhë programuese e nivelit të ulët, por më e lexueshme se numrat binarë. Si assembler nihet programi kompjuterik i cili mund të kuptojë një reprezentim të programit që është i lexueshëm nga programeri, dhe e përkthen këtë kod në reprezentimin binar.

Assembly Language është një gjuhë programuese, që sot nihet si gjuha më e ulët e makinës kompjuterike, dhe shpeshherë shkurtohet në “asm”. Konsiderohet e nivelit të ulët pasiqë cdo funksionalitet i cili është prezent, përfaqëson një instruksion të vetë procesorit. Pasiqë mvaret nga arkitektura e procesorit, Assembly ka dialekte për secilen arkitekture të procesorit, që do të thotë se secili lloj i procesorit e ka verzionin e vetë të kesaj gjuhe. Programi i cili e përkthen kodin e shënuar në assembly, nihet si assembler.

**Shembulli për Assembly**

```
ection .text
    global_start      ; per linker

_start:               ; sikur main() ne java apo c
    mov edx,gjatesia  ; gjatesia e mesazhit
    mov ecx,msg       ; mesazhi qe do te shkruhet
    mov ebx,1         ; lloji i file-it ( output stream )
    mov eax,4         ; nje syscall per te shkruar mesazhin ne ekran
    int 0x80          ; thir kernel

    mov eax,1         ; system call per te dalur nga programi
    int 0x80          ; thir kernel

section .data
msg db 'Pershendetje', 0xa   ;stringu qe do te printohet
gjatesia equ \$ - msg        ;gjatesia e stringut
```

Në kodin e mësipërm është paraqitur një Assembly program i cili printon “Pershendetje” në ekran. Edhe si shihet mësipërm, Assembly është një gjuhë që kërkon dijeni intrike të arkitektures së procesorit dhe hardware-it të makines. Përndryshe nga gjuhet e nivelit të lartë, nuk ka struktura dinamike dhe është shumë e vështirë të lexohet, mirëpo shpejtësia e ekzekutimit të programit të shkruar në Assembly është më e shpejtë se cdo gjuhë programuese tjetër. Kompajlleret si CLang, GCC dhe ai i Microsoftit për C dhe C++ janë në gjendje të gjenerojnë Assembly kod për programet e kompajlluara, dhe për programerët e aftë kjo është një aftesi e pa krahasueshme. Edhpse më i lexueshëm dhe më i programueshëm se sinjalet binare, Assembly është një gjuhë që kërkon trajnim dhe eksperiencë, dhe përvec kësaj është një gjuhë që nuk preferohet për programet e mëdha dhe komplekse, pasiqë kodimi joadekuat mund të shkaktojë “bugs”. Këto ishin arsyet se pse u investua më shumë në teknologjinë e kompajllerave. Kështu edhe fillon epoka e re e gjuheve të “nivelit të lartë”, në vitet 1950 me gjuhët si Short Code, Autocode, Fortran etj. Secila ishte e specializuar dhe kishte qëllimin e vetë, që zakonisht kishte të bëj me reprezentimin e formulave shkencore dhe matematikore. Falë këtyre specializimeve, janë bërë shumë investime teknologjike jo vetëm të kompajllereve por edhe kompjutereve.

## 1.2 Programet në nivelin e hardware-it

Secila pajisje elektronike, qoftë komplekse apo e thjeshtë, në nivelin e hardware-it manipulon sinjalet elektrike për të aritur efektin e dëshiruar. Kur krijojmë programe kompjuterike, apo software, ne në menyrë indirekte manipulojmë këto sinjale në nivelin fizik ashtuqë makina kompjuterike na jep efektin të cilin e kerkojmë. Këto teknika mundesohen nga pajisje të thjeshta që bazohen në voltazhe të rrymes të njohura si tranzistorë.

### 1.2.1 Tranzistoret dhe Qarqet Logjike

Tranzistori mund të përshkruhet si një rezistor vlera e të cilit ndryshon në bazë të voltazhes, dhe kjo aftësi e këtyre pajisjeve mundëson që në menyrë perfekte të përshkruhen sinjalet binare 0 dhe 1. Në rrast se kemi 0v, tranzistori nuk aktivizohet fare, ndërsa nëse kemi më shumë se 3.5v, ateherë tranzistori lejon kalimin e elektroneve. Mund ta marrim shembullin e një poqi elektrik, ku nëse nuk ka rrymë nuk do të funksionoj, mirëpo nëse qarku elektrik mbyllet atëherë poqi elektrik hapet. Tranzistoret janë vetëm faza e parë. Me kombinimin e tranzistoreve mund të krijojmë qarqet logjike ( logic gates ), që janë qarqe elektrike të cilat implementojnë logjikën e bulit apo boolean logic. Edhpse kemi disa lloje të qarqeve logjike si AND, OR, XOR, NOT, të cilat kanë funksionalitete specifike, fakti që të gjitha këto lloje të qarqeve duhej të implemtoheshin në një formë kompakte nuk ishte zgjidhje. [2]Shkencëtari Charles Sanders Peirce ka vërtetuar se qarqet NAND dhe NOR ishin në gjendje që të reprezentonin të gjitha qarqet tjera.

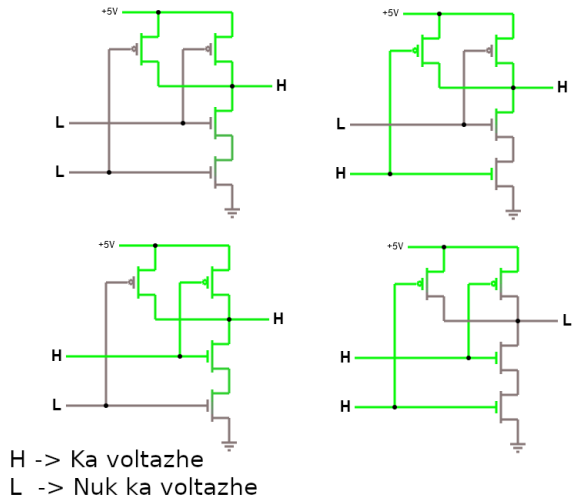


Figura 1.1: NAND qarqet dhe kontrollimi i sinjaleve

Në figurë mund të shohim demonstrimin e NAND qarkut, i cili është në gjendje të reprezentoj të gjitha qarqet logjike që përdoren nga makina kompjuterike. Tranzistorët krijojnë qarqet logjike, dhe këto qarqe janë blloqet e ndërtimit të cdo sistemi digjital, duke përfshirë edhe kompjuteret.

### 1.2.2 Gjuha e Makines

Pasi një sistem digjital sikur makina kompjuterike është e ndërtuar me qarqe që mund ti kuptojnë sinjalet binare 0 dhe 1, edhe vetë sistemi kompjuterik i kupton dhe manipulon këto sinjale. [3]Gjuha e makines është një koleksion i biteve që një sistem kompjuterik mund ti identifikojë dhe të kryej një veprim përkatës. Në një sistem kompjuterik pajisja që e ka gjuhën e vetë nihet si CPU, apo ndryshe Central Processing Unit (procesori). Procesori është në gjendje të identifikojë dhe ekzekutojë një seri të biteve që mund të thiren si Instruksionet e Makinës. Pikerisht këto instruksione të kombinuara në forma të ndryshme nihen si "gjuha e makinës". Lind pyetja se përse këto seri të biteve shkaktojnë një veprim përkatës? Arsyeja është se përfundi secilit instruksion egziston një qark i cili e implementon këtë instruksion. Do të thotë se gjuha e makines në fakt është vetëm kontrollimi i sinjaleve elektrike përmes qarqeve logjike të

implementuara në hardware, dhe programet e shkruara në Assembly, C, Java ose ndonjë gjuhë tjeter, janë vetëm abstraksione të nivelit të lartë .

### 1.2.3 Gjuhët e niveleve më të larta

Sikurese u përmend edhe më siper, Assembly është një reprezentim simbolik i gjuhës së makinës ( numbrave binare ) dhe kjo u implementua duke i emërtuar bitet të cilat kryejnë një instruksion. Ky lloj i programimit ishte një përmisim, sepse programeret nuk kishin nevojë të shkruanin numrat binar por vetëm emrat e instruksioneve. Kështu përshembull në vend se të shënohej 000000101, to të përdorej ”hlt”. Një abstraksion si ky ishte më lehtë që të memorizohej dhe si pasojë programet dhe kodet filluan të krijoheshin më shpejtë.

Kur shkruajmë kod në Assemby dhe e ruajmë në një file, programi i cili e përpunon këtë file nihet si "assembler"dhe detyra e assembler-it është që ta përkthejë këtë file në vlerat binare që sistemi kompjuterik mund ti kuptojë. Mirëpo lind pyetja, si është e mundur që ky assembler të egzistojë? Të imagjinojmë për një moment që ky assembler është i pari në botë, ateherë ky program u krijua me numra binar. Më vonë, pas krijimit të assembler-it të parë, krijohet një assembler i ri me Assembly, dhe përdoret verzioni i parë ( i krijuar me numra binarë ) për ta kompajlluar për makinën. Këto vlera binare më vonë mbahen në një pajisje tjeter mbrenda kompjuterit e cila i dërgon këto një dekoderi. Kjo është e vërtetë edhe për gjuhet e nivelit më të lartë të programimit! Si shembull mund ta marrim gjuhën programuese C, versioni i parë (kompajlleri) i të cilit u krijua me assembly dhe pasi që ky kompajller ariti një pjekuri, u përdor për ta krijuar verzionin e ri të vetë-vetes me C. Edhe Java, një gjuhë tjeter programuese e nivelit të larte fillimisht u implementa me C dhe C++, mirëpo verzionet e ardhshme të Java-s u krijuan me Java. Procesi i krijimit të një kompajlleri, i cili më vonë është në gjendje të kompajllojë vetë-veten nihet si [4]bootstraping.

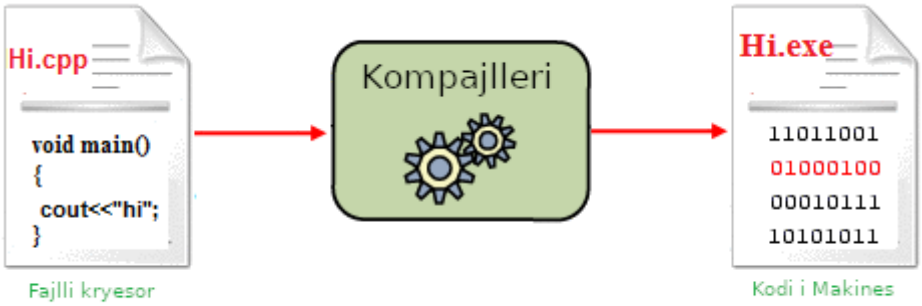


Figura 1.2: Mund të shihet procesi i kompajllimit nga një nivel i lartë

# Kapitulli 2

# Terminilogjia dhe Teknologjia

Në këtë kapitull do të sqarojmë linguistikën që i rrethon gjuhët programuese dhe teknologjitë e tyre, ashtuqë kapitujt e ardhshëm të jenë më të kupueshëm. Më poshtë gjinden sqarimet e fjalëve dhe teknologjive që lidhen në një formë apo tjetër, me këtë punim dhe janë të rëndesishme në kapitujt e ardhshëm. Përveq kompajllerit ( teknologjia e cila duhet të sqarohet për të qartësuar pjesët tjera ), ALP implementon të gjitha sistemet tjera!

## 2.1 Source Code

[5]Source Code apo përkthyer në Shqip kodi burimor është një listë e udhëzimeve që një programer i shkruan në një tekst editor ( notepad apo ndonjë program tjetër ). Ky kod, varësisht se në cilën gjuhë programuese është shkruar, i jipet një kompajlleri apo interpretuesi dhe nëse kodi i shkruar në file ka kuptim dhe nuk ka gabime sintaktike, atëherë e kthen një rezultat. Source Code është burimi i një programi kompjuterik. Përmban deklarime, udhëzime, funksione dhe deklarata të llojeve të ndryshme, të cilat përmbajnë udhëzime për ekzekutim në sistem. Një program mund të permbajë një apo më shumë source code file-a, dhe këto file-a ruhen në sistemin kompjuterik. Source code-i përmban edhe komente të cilat nuk ekzekutohen por injorohen, dhe komentet përdroren për komentimin e sourec code-it apo ndonjë programi.

```
klasa ALP {
    Pershendetje( emri ){
        shkruaj( "Pershendetje " + emri + ":)" );
    }
    Test(){
        per( deklaro i=0; i<10; i=i+1 ){
            ky.Pershendetje( "Alixhan Basha" );
        }
    }
}

deklaro a = ALP();
a.Pershendetje( "Alixhan Basha" );
a.Test();
// perfundimi
```

Kodi 2.1: Si duket kodi i shkruar në ALP

## 2.2 Kompajlleri

[6] Kompajlleri është një program softverik i cili është në gjendje të transformojë source code-in e shkruar në një gjuhë programuese nga një programer, në gjuhën e makinës (kodin binar). Procesi i konvertimit të një gjuhe programuese të nivelit të lartë ( Java, C, Rust, etj ) në kodin binar të cilin e kupton procesori i kompjuterit nihet si kompajllim. Ka lloje të ndryshme të kompajllereve që kryejnë funksione të ndryshme, për shembull një kompajller i cili e përkthen kodin e shënuar për një arkitekturë apo sistem operativ ndryshe nga ai ku gjindet nihet si cross-compiler. Një bootstrap kompajller është një kompajller i shkruar në gjuhën programuese të cilin e kompajllon (vetë-veten).

Krijimi i kompajllereve ndahet në tre faza unike. këto faza nihen si Front End, Middle End dhe Back End. Secila ka funksionin e vetë dhe arsyen se përse përdoret.

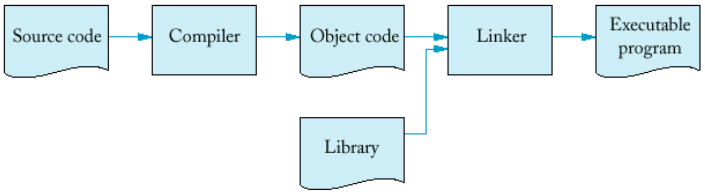


Figura 2.1: Fazat e kompajllerit

[7]Front End i kompajllerit është pjesa e cila skanon dhe verifikon sintaksën dhe aspektin semantik të një gjuhë specifike, për shembull C++. Në rrastet kur programi i shkruar ka gabime në sintakse atëherë kjo fazë e kompajllimit duhet të raportoj eroret në mënyre të saktë. Në pjesën e Fron End-it futet analiza statike, analiza leksikale, dhe analiza semantike.

Middle End zakonisht miret me optimizim e kodit. Në rrastet kur një gjuhë programuese ka një gjuhë mesatare ( Java Bytecode, Python Bytecode etj ) optimizimet bëhen në kodin mesatar. Optimizimi i kodit zakonish bëhet në pjesët e kodit të panevojshëm, kodit të pa-aritshëm dhe mund të aplikohet edhe në shumë pjese të ndryshme të kompajllerit.

Back End merr kodin e optimizuar nga hapi i mëparshëm. Edhe këtu, nëse është e nevojshme bëhen kontrollime të kodit me analiza apo edhe transformime. Back End po ashtu e krijon kodin në bazë të arkitekturës së sistemit, dhe merret me detajet e nivelit të ulët.

## 2.3 Interpretuesi

[6] Interpretuesi është një program kompjuterik për ekzekutim të kodit të një gjuhe programuese. Eshtë tejet e ngjajshme me kompjallerët si në praktikë ashtu edhe në implementim. Mirëpo dallimi kryesor i komapjllerit dhe interpretursit është se një gjuhë programuese e interpretuar nuk përkthehet në kodin binar përmes kompajllimit. Një gjuhë programuese e interpretuar ekzekutohet aty për aty, në momentin që interpretuesi thiret. Gjuha që do të implementohet përgjatë këtij punimi është një gjuhë e interpretuar. Si gjuhët më të njohura që perdorin teknologjinë e interpretuesve janë Python, JavaScript, Perl, BASIC, MATLAB, etj. Zakonisht gjuhët e interpretuara nuk kanë datatipe strikte, kjo do të thotë se nuk egziston "type checking"dhe cdo variabël të definuar mund ti ndërhoet tipi.

Duke pasur parasysh faktin që një gjuhë e kompajlluar e përkthen dhe e optimizon source code-in e shkruar në një gjuhë si C në kodin binar që e kupton makina, interpretuesi përkthen, optimizon dhe ekzekuton kodin rresht-pas-rreshti cdo herë që thirret, së bashku me një "Run-time"më të madh dhe tendnecën për të bërë më shumë me më pak rreshta kod shkakton që një interpretues të jetë më i ngadaltë se një kompajller. Mirëpo kjo mund të mvaret edhe nga implementimi i këtyre teknologjive.

## 2.4 Gjuhët dinamike dhe statike

Termet dinamike dhe statike në kontekstin e gjuhëve programuese u referohen datatipeve të kësaj gjuhe. Termi dinamik do të thotë se gjuha programuese dhe dataipet janë dinamike, në atë formë që nuk ka tipe të sakta. Përshembull një variabël e definuar mund të jetë një numer i plotë ( integjer ) në momentin e deklarimit, mirëpo më vonë në kod mund të ridefinohet si një objekt, string apo numer dhjetor ( double ). Ndersa termi "statik"do të thotë që gjuha programuese ka rregulla të sakta lidhur me menyrën se si reprezentohen variablat. Përshembull Java, C, C++, Rust janë gjuhë programuese statike. Nëse një variabël definohet si numër i plotë, do të qëndrojë si numër i plotë derisa ajo variabël të shkatërohet apo derisa programi të mbyllet.

## 2.5 Lekseri - Skaneri

[8] Hapi i parë për një interpretues apo kompajller është skanimi. Skaneri apo ndryshe "Lexer"si input mer source code-in dhe e kthen atë në një grupim të copëzave të quajtura tokena. Këto janë "fjalët"dhe "pikësimet"të cilat kanë kuptim dhe që përbëjnë gramatikën e gjuhës. Pra skaneri zgjidh problemin e njohjes nëse kodi i dhënë është anëtar i ndonjë gjuhe. Skaneri thjesht e kthen tekstin e pakuptimtë në një listë të tokeneve si "numër", "string", "identifikues"ose "operator"dhe mund të bëjë gjëra të tilla si njohjen e fjaleve kyqe (keywords) dhe heqja e hapësirës së bardhë. Një skaner njeh disa grupe gjuhësh të rregullta. Një gjuhë ë rregullt"është ajo që mund të analizohet pa ndonjë gjendje shtesë. Kjo do të thotë se është shumë efikase. Duhet të shikohet vetëm një bajt për të marrë vendime, dhe të gjitha vendimet mund të paketohen në një matricë vendimesh të quajtur "Finite Autometa".

```
alp: deklaro nje_variabel = "Alixhan";
Tokenat e krijuar nga Skaneri/Lekseri
< [VAR] [deklaro] [deklaro] >
< [IDENTIFIER] [nje_variabel] [nje_variabel] >
< [EQ] [=] [null] >
< [TEXT] ["Alixhan"] [Alixhan] >
< [SEMICOLON] [;] [null] >
```

Figura 2.2: Tokenat e gjeneruar nga ALP-Lexer

Një aspekt i rëndësishëm i skanerit është trajtimi i hapësirave të bardha dhe komenteve. Në shumicën e gjuhëve, semantika e gjuhës është e pavarur nga hapësira e bardhë. Hapësira e bardhë kërkohet vetëm për të shënuar fundin e një tokeni dhe ka pak vlerë në hapat e ardhshëm. Sidoqoftë, ky nuk është rasti me çdo gjuhë, sepse hapësira e bardhë mund të këtë kuptime semantike në disa gjuhë siç është Python. Skaneret e ndryshëm trajtojnë ndryshe këto hapësira të bardha dhe komente.

## 2.6 Parseri

Parseri është përgjegjës për leximin e tokeneve nga skaneri, dhe prodhimin e AST(Abstract Syntax Tree). Disa e quajnë këtë fazë si skaneri i dytë, pasiqë e vazhdon punën aty ku e ka lënë skaneri. Parseri merr tokenin e ardhshëm nga skaneri, e analizon atë dhe e krahason me një gramatikë të përcaktuar. Pastaj vendos se cila nga rregullat gramatikore duhet të merren parasysh, dhe vazhdon analizën sipas gramatikës. Mirëpo, kjo nuk është gjithmonë kaq e drejtpërdrejtë, pasi ndonjëherë nuk është e mundur të përcaktohet rruga që duhet ta ndjek vetëm duke parë tokenin e ardhshëm. Kështu, parseret duhet të kontrollojnë disa tokena më përpara, për të vendosur rrugën ose rregullin gramatikor që do të merret parasysh. Në mënyrë që kodi i shkruar në formë të lexueshme nga njerëzit të kuptohet nga një makinë, ai duhet të shëndrrohet në gjuhën e makinës. Kjo detyrë zakonisht kryhet nga interpetuesi apo kompajlleri. Parseri zakonisht përdoret si një përbërës i interpretuesit apo kompajllerit që organizon source code-in në një strukturë që mund të manipulohet lehtësisht (AST). ALP do të implementoj një lloj të parserit të njohur si ”Recursive Descent Parser”.

## 2.7 Abstract Syntax Tree

[9] Në shkencat kompjuterike, një Abstract Syntax Tree (AST), ose thjesht pema sintaksore, është një reprezentim i sintakses së kodit burimor të shkruar në një gjuhë programuese. Secila nyje e pemës reprezenton një konstrukt që ndodhet në kodin burimor. Pemët sintaksore abstrakte janë struktura të dhënash të përdorura gjërësisht në kompajllera për të përfaqësuar strukturën e kodit të programit. Shpesh shërben si një përfaqësim i ndërmjetëm i programit përmes disa fazave që kërkon kompajlleri, dhe ka një ndikim të fortë në prodhimin përfundimtar të kompajllerit. Në procesin e përpunimit të një programi burimor në kodin e synuar, një kompajller mund të ndërtojë një ose më shumë përfaqësime të ndërmjetme, që mund të kenë forma të ndryshme. Pemët sintaksore ose AST janë një formë e përfaqësimit të ndërmjetëm, ato krijohen dhe përdoren zakonisht gjatë analizës sintaksore dhe semantike. Pas sintaksës dhe analizës semantike të programit burimor, shumë kompajllerë krijojnë një përfaqësim të nivelit të ulët ose të ngjashëm me makinën, i cili mund të mendohet si një program për një makinë abstrakte. Kjo paraqitje e ndërmjetme duhet të këtë dy veti të rëndësishme: duhet të jetë e lehtë për tu prodhuar dhe duhet të jetë e lehtë për tu përkthyer në makinerinë e synuar.

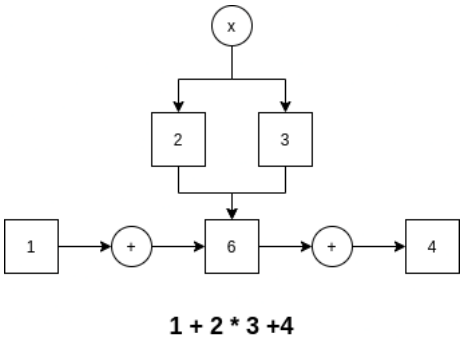


Figura 2.3: një AST për një kod të thjeshte

## 2.8 Analiza Statike

Analiza e kodit statik dhe analiza statike shpesh përdoren në mënyrë të ndërlidhur, së bashku me analizën e kodit burimor. Ky lloj i analizës u referohet dobësive në source code që mund të shkaktojnë rezultate të padëshiruara apo bugs-a. Kjo mund të arrihet edhe përmes rishikimeve manuale të kodit. Por përdorimi i mjeteve të automatizuara është shumë më efektive. Një analizuas statik përdor pemën e gjeneruar nga parseri dhe është në gjendje të analizojë deklarimet e një gjuhe programuese dhe të raportoj në rraste kur gjen diqka që mund të shkaktojë dështime gjatë ekzekutimit. Përndryshe nga interpretuesi apo kompajlleri, një analizues statik nuk e ekzekuton kodin, vetëm e analizon ate.

## 2.9 Runtime

Runtime është faza përfundimtare e ciklit jetësor të programit në të cilën makineria ekzekuton kodin e programit. Parase të fillojë ekzekutimin një program në sistemin kompjuterik, alokohet memorja, lidhen libraritë ( library linking, kodet tjera ) dhe fillon ekzekutimi. Përderisa programi ekzekutohet në kompjuter, ky proces nihet si ”Runtime” apo koha e ekzekutimit. Gjatë runtime mund të shfaqen errora të ndryshme, kështu që një gjuhë programuese duhet të jetë në gjendje të raportoj cdo lloj defekti gjatë kësaj fazës. Përveq runtime kemi edhe ”Compile-time", e cila reprezenton kohën gjatë së cilës kompajlleri është aktiv dhe kryen detyrën e vetë me source code-in e dhënë. Zakonisht gjuhët e niveleve të larta si Java, Python, JavaScript kanë edhe Runtime-a të mëdha, që shtojnë kompleksitet dhe ngadalsojnë gjuhën, mirëpo për shkak se runtime-i është i madh dhe kompleks poashtu ka edhe funksionalitet të lartë, dhe mund të jetë një aspekt pozitiv për programerin.

# Kapitulli 3

# ALP - Gjuha Programuese

Në këtë kapitull do të flitet për ALP ( ALbanian Programming language ). Gjithqka duke filluar nga definimi, implementimi ( pjesët e mbrendshme të interpretuesit ) dhe testimi. Përmes fotografive të kodit do të arijmë të kuptojmë sintaksën dhe konceptet e ndryshme që mund të reprezentohen me ALP.

## 3.1 Definimi i ALP

ALP është një gjuhë programuese dinamike e orientuar në objekte. Variablat mund të ruajnë vlera të çdo lloji, dhe madje një variabël mund të ruaj vlera të llojeve të ndryshme në kohë të ndryshme. Nëse mundohemi të kryejmë një operacion me vlera të llojit të gabuar - të themi, pjestimi i një numri me një tekst, atëherë gabimi zbulohet dhe raportohet gjatë kohës së ekzekutimit. Përveq kësaj, ALP do të ketë sintaksen në gjuhën Shqipe, ndërsa semantika do të jetë e ngjajshme me Java dhe Javascript. Si inspirim për dizajnin e gramatikës dhe rregullave të ALP kanë qenë gjuhet programuese Python, Java dhe Javascript.

### 3.1.1 Datatipet

Përdorimi i varaiblave është një kërkesë bazike për një gjuhë programuese. Edhe gjuhët më të thjeshta kanë mënyra për rruajtjen e vlerave në memorje. Edhepse ALP është dinamike, në Runtime duhet të kemi një ide se cfarë datatipi ka një variabël, në menyrë që të lejojmë operime të caktuara. Përshembull mateamtika me numra duhet të lejohet, ndersa pjestimi me stringje jo. Datatipet që janë prezente në ALP,dhe ato që i kupton interpretuesi janë.

- Boolean: ALP ka dy reprezentime të datatipit boolean: vertet dhe fals. Këto mund të përdoren për të programuar ose reprezentuar gjëra të ndryshme.
- Numër Integjer: Datatipi i cili reprezenton numrat e plotë pozitiv dhe negativ. Mund të mblidhet, zbritet, shumëzohet dhe pjestohet. Diqka që është e mundshme në gjuhet si Java apo JavaScript është mbledhja e numrave me stringje, ku rezultati i kthyer është një string që permban numrin. Edhe ALP e ka këtë aftësi.
- Numër Dhjetor: Datatipi i cili reprezenton numrat dhjetorë pozitiv dhe negativ. Rregullat e lartepermendura vlejnë edhe për këtë datatip.
- String apo Tekst: Datatipi për reprezentimin e shkronjave, fjalëve apo edhe teksteve,

suporton edhe karakterët si \n, \t etj. Stringjet mund të mblidhen ( bashkohen ) në mes veti dhe me numra. Operacionet tjera matematikore do të shkaktojnë errore.

- Null apo Zbrazet: Kur dëshirojmë të reprezentojmë vlerat joegzistuese. Me datatipin "ZBRAZET"ose "NULL"nuk mund të kryhen kurfare operime matematikore.

Këto të lartepermendurat janë datatipet primitive të ALP, ndërsa kemi edhe tipet komplekse si Objektet, Klasat dhe Funksionet. Në gjuhet statike të kompajlluara si Java, funksionet nuk janë datatipe dhe vetem mbajne një grupim të instruksioneve që ekzekutohen kur thiren, ndersa në ALP një funksion jo vetem që ka funksionalitet të lartëpërmendur, por edhe mund të përdoret në formë të 'Closure'. Do të flitet në më shumë detaje për këtë pjesë të gjuhes më poshtë.

### 3.1.2 Expressions - Shprehjet Matematikore dhe Programatike

Nëse datatipet e një gjuhe programuese janë atomet, Expression-sat apo shprehjet janë molekulat e saj. Detyra e një shprehje në ALP është të prodhojë një vlerë. Sikurese në matematike, edhe sikurese pritet prej një gjuhe programuese ALP mbështe të gjitha llojet e shprehjeve matematikore. Eshtë "Turing Complete".

- Aritmetika: ALP mbështet të gjitha operatoret aritmetik bazik, që gjinden në C dhe gjuhet e bazuara në ate. Operatoret si +,-,*,/ dhe %.

**Aritmetika në ALP**

```
deklaro n1 = 10; // deklarimi dhe inicializimi i variables 'n1'
deklaro n2 = 5 ; // deklarimi dhe inicializimi i variables 'n2'

deklaro rezultati = n1 + n2; // mbledhje
        rezultati = n1 - n2; // zbritje
        rezultati = n1 * n2; // shumezim
        rezultati = n1 / n2; // pjestim
        rezultati = n1 % n2; // mbetja
```

- Krahasimi dhe Barazimi: Operatorët që gjethmonë kthejnë një vlere boolean. Kuptohet edhe vetë nga emri se në këtë grup të operatoreve bëjnë pjesë ato të krahasimit dhe të barazimit.

**Operatoret për Karhasim dhe Barazim në ALP**

```
// Vazhdimi nga ALP kodi i mesiperm
shkruajr( n1 <  n2 ); // fals
shkruajr( n1 <= n2 ); // fals
shkruajr( n1 >  n2 ); // vertet
shkruajr( n1 >= n2 ); // vertet
shkruajr( n1 == n2 ); // fals
```

- Operatorët Logjik: në këtë grup futen operatoret si "!","dhe","ose". Operatori "!" shkakton negacion të vlerave boolean. Operatori "dhe" mundëson ekzekutimin e komandës së ardhshme vetëm nëse komanda e mëparshme ka pasur sukses. Operatori "ose" ekzekuton komandën e parë apo nëse dështon, komanden e dyte apo të treten, e kështu me rradhë.

**Operator logjik në ALP**

```
// Vazhdimi nga ALP kodi i mesiperm
shkruajr( !vertet ); // fals
(1==1) dhe shkruajr( "OK" ); //1 eshte 1, keshtuqe ekzekuton
                             //komanden e rradhes

(1==2) dhe shkruajr( "OK" ); //1 nuk eshte 2, keshtuqe
                             //ekzekutimi nuk vazhdon

(1==1) ose shkruajr( "OK" ); //1 eshte 1, keshtuqe nuk e
                             //ekzekuton komanden e rradhes

(1==2) ose shkruajr( "OK" ); //1 nuk eshte 2, keshtuqe
                             //kontrollon komanden tjeter
```

- Përparësia dhe Grupimi: Kushtet e mësiperme funksionojnë ashtu se si pritet nga një gjuhë programuese. Aritmetika i përmbahet rregullave të matematikës, ashtuqë në rrasin '1+2*3' fillimisht ka përparësi shumezimi dhe pjestimi, e me pas mbledhja dhe zbritja.

### 3.1.3 Statements - Deklarimiet Programatike

Në ALP një deklarim përbëhet nga një apo më shumë shprehje (Expression-e). Sikurse e përmendëm më sipër, detyra e një "Expression" është që të prodhojë një vlerë, detyra e një deklarimi është që të prodhojë një efekt anësor. Si pasojë e dizajnit, deklarimet nuk prodhojnë një vlerë, e për të qenë funksionale ato duhet të prodhojnë një efekt. Të marim shembullin e meposhtëm:

**Disa deklarime programore të ALP**

```
//deklarim qe ndron gjendjen globale te interpretuesit
importo "<Standard>";
shkruajr("Hello World");
```

E shohim deklarimin për importim të një moduli për ALP. Deklarimi 'importo' ndyshon gjendjen globale të interpretuesit pasiqë është në gjendje që të ekzekutojë kod nga një file tjetër i cili mund të gjindet kudo në kompjuterin e programuesit. Mund ta shohim funksionin ekuivalent në Python si 'print'. Funksioni 'shkruajr' mer si parametër një shprehje 'String literal expression' dhe e afishon ate në ekran. Përmes simbolit për terminim ";" një shprehje e thjeshtë kthehet në një deklarim.

### 3.1.4 Variablat

Variablat përdoren për të ruajtur informacione, ku më pas mund të i referohemi dhe mund ti manipulojmë. Ato gjithashtu mundësojne etiketimin e të dhënave me një emër përshkrues, kështu që programet mund të kuptohen më qartë nga lexuesi dhe nga vetja. Eshtë e dobishme që variablat të mendohen si kontejnerë që mbajnë informacion. Qëllimi i tyre i vetëm është etiketimi dhe ruajtja e të dhënave në memorje. Në ALP variablat mund të deklarohen me fjalen kyqe ”deklaro”. në rrast se variabla nuk inicializohet nga programeri, atëherë vlera e variables se sapokrijuar do të jetë ’ZBRAZET’. Pas deklarimit të variablës, ajo mund të përditësohet, ndyshohet vlera apo edhe mund të ju ndryshohet edhe tipi. Po ashtu variablat do të kenë një ’scope’ apo fushëveprim të vetin. Kjo do të thotë se në ALP kemi suport për variabla globale dhe lokale.

**Fushëveprimi definimi dhe deklarimi i variablave në ALP**

```
deklaro _null; // Do te inicializohet si ZBRAZET
deklaro emri = "Alixhan";
deklaro mosha = 21;

emri = 100.3; // kur u deklarua ishte string, tani int
mosha = "hello"; // ishte int, tani string
{
    // nje fusheveprim i ri, i cili ka akses ne fusheveprimin
    // e siperm, por gjithqka qe deklarohet ketu
    // shkaterohet pasi qe dalim nga ky fusheveprim

    deklaro _do_te_shkatrohet = " nje vlere ";
    shkruajr( emri ); // OK
    shkruajr( _do_te_shkatrohet ); // nje vlere
    // ...
}
shkruajr( _do_te_shkatrohet ); // error: variabla nuk egziston
```

### 3.1.5 Kontrollimi i ekzekutimit

Eshtë vështirë të shkruhen programe të dobishme nëse nuk mund të tejkalohen disa pjesë të kodit ose të ekzekutohen disa më shumë se një herë. Pikerisht edhe këtu futet kontrollimi i ekzekutimit të kodit. Cka do të ekzekutohet në bazë të disa kushteve, sa herë do të ekzekutohet një komandë? Të gjitha këto pergjigje gjinden në kontrollim të ekzekutimit. Përveç operatorëve logjikë që kemi mbuluar tashmë, ALP trashëgon tre shprehje (Expression) drejtpërdrejt nga C. Ato janë: if, else if, else, for dhe while apo si paraqiten në ALP: ”nese”, ”tjeter”, ”perndryshe”, ”per” dhe ”perderisa”. Këto janë konstrukte që mundesojnë ekzekutim të kodit në bazë të ndonjë kushti, përseritja e saj n-herë dhe së bashku me operatorët logjik mund të krijohen kushte të shumefishta dhe programe komplekse.

**If/Else deklarimet në ALP**

```
deklaro numri = 1;

nese( numri == 1 ){ /* ... deklarimet e ndryshme */ }
tjeter( numri == 2 ose numri == 3 ) {
    // ... deklarimet e ndryshme
}
perndryshe { /* ... deklarimet e ndryshme */ }

per( deklaro i=0; i<10; i=i+1 ){ /* for loop */ }

perderisa( numri < 10 ){
    // while loop
    numri = numri+1;
}
```

Më sipër mund të shihen deklarimet për kontrollim të ekzekutimit të kodit në ALP. Secila prej tyre ka detyrën dhe funksionin e vetë. Blloku ”nese” pranon një seri të shprehjeve( Expression ) dhe nëse ato shprehje kthejnë rezultatin ”vertet” atëherë ekzekutohet trupi, nëse nuk kthehet rezultati vërtet dhe nëse ka ”tjeter” ose ”perndryshe”, atëherë do të vleresohen edhe ato blloqe. në momentin që egziston ”nese” dhe ”tjeter”, edhe kushti nuk plotësohet, ateher blloku ”perndryshe” do të ekzekutohet.

Blloku ”per” mer një deklarim dhe dy shprehje. Përderisa shprehja e kushtit është e vërtetë, blloku i shprehjes ”per” do të përseritet. Ideja është e njejtë edhe me deklarimin ”perserit”. Kllapat gjarpërore sinjalizojnë fillimin dhe mbarimin e fushëveprimit të trupit të deklarimeve. Secila ka qasje në fushëveprimet ”e siperme” dhe cdogjë e krijuar mbrenda bllokut shkatërohet kur dalim nga blloku.

### 3.1.6 Funksionet

Një funksion është një bllok kodi që kryen një detyrë. Mund të thirret dhe ripërdoret shumë herë. Programeri mund të jep informacionet një funksioni dhe funksioni mund të kthejë informacione të ndryshme. Shumë gjuhë programuese kanë funksione të integruara që mund të përdoren në libraitë e tyre, por gjithashtu programeri mund të krijojë vetë funksionet e veta. Funksionet, si duken dhe si thiren janë identike si në JavaScript. Sikurese variablat, programeri mund të i definojë funksioniet e veta dhe ti krijoj ashtu se si dëshiron ( përderisa source code-i përputhet me rregullat e ALP ), dhe nëse gjithqka është në rregull atëherë programi ekzekutohet pa problem. Secili funksion e ka fushëveprimin e vetë që sinjalizohen përmes kllapave gjarpërore. Gjithcka që është mbrenda këtyre kllapave përbën trupin e funksionit. Në momentin që thirret një funksion, cfarëdo deklarimi që është në trup ekzekutohet. Funksionet në ALP krijohen me fjalën kyqe ’funksioni’.

**Definimi dhe thirja e funksioneve në ALP**

```
funksioni Mesashi( diqka ){
    nese( diqka != ZBRAZET ){
        shkruajr( diqka );
    }
}
Mesazhi( "Hello World!" );
```

Edhe se si shihet më siper, funksionet në fakt mundësojnë që kodi i shënuar një here, të mos përseritet. Në rrast se definimi i një funksioni përmban argumente, dhe kur thirret funksioni për ekzekutim nuk ja japim argumentet e pritura, ALP do të shfaq një error duke thëne se "Printen n argumente, n janë dhënë". Eshtë e rëndësishme të kuptohen disa terme lidhur me funksionet. Fjalët parametër dhe argument për shumicën e programereve janë e njejta gjë, mirëpo për nga implementimi dhe se ku përdoren kanë dallime.

- Argumentet: Argumenti është vlera aktuale që i jipet një funksioni kur ai thirret. Pra, një thirrje e funksionit ka një listë argumentesh.
- Parametrat: Një parametër është një variabel që mban vlerën e argumentit brenda trupit të funksionit. Kështu, një deklarim i funksionit ka një listë të parametrave.

### 3.1.7 Closures - Funksionet

Një aftësi e ALP, e cila është inspiruar nga gjuhët funksionale dhe JavaScript është ”Closures”. Këto janë funksione mbrenda funskioneve të cilat e rruajnë gjendjen e rrethit që e kanë. Në JavaScript, në mund ta kthejme një funksion nga mbrenda një funksioni egzistues, të njejten gjë mund ta bejmë edhe në ALP.

**Closure Funksionet në ALP**

```
funksioni Test( diqka ){
    nese( diqka != ZBRAZET ){ shkruajr( diqka ); }
    funksioni TestMbrendaTest(){
        shkruajr( "test mbrenda test" );
    }
    kthen TestMbrendaTest;
}
Test( "Hello World!" );
```

Më sipër mund ta vërejmë se funksionet mund të na kthejnë funksione dhe këto nryshe nihen si ”Closures”. Përdorimi i Closure-ave shoqërohet në gjuhët programuese ku funksionet janë objekte të klasit të parë, në të cilat funksionet mund të kthehen si rezultate të funksioneve të rendit më të lartë, ose të pasohen si argumente të funksioneve të tjera. Nëse funksionet me variablat e lira janë të klasit të parë, atëherë kthimi i tyre krijon një Closure. Kjo përfshin edhe functional programming languages si Lisp dhe ML, si dhe shumë gjuhët moderne si Python dhe Rust. Closure-at përdoren shpesh me funksionet ”callback”, veçanërisht për rrast të një event-i, për shembull në JavaScript, ku përdoren për ndërveprime me një faqe dinamike në web. Konstruktet si objektet dhe strukturat e kontrollit mund të progamohen me një Closure. Në disa gjuhë programuese, një Closure mund të shfaqet kur një funksion definohet mbrenda

një funksioni tjetër, dhe funksioni i mbrendshëm i referohet variablave lokale të funksionit të jashtëm. Në runtime, kur ekzekutohet funksioni i jashtëm, formohet një Closure, i përbërë nga kodi i funksionit të mbrendshëm dhe referencat (vlerat e variablave) për çdo variabël të funksionit të jashtëm të kërkuar nga mbyllja. Meqenëse Closure-at vonojnë vlerësimin - d.m.th., ato nuk ”bëjnë” asgjë deri në momentin që thirren, ato mund të përdoren për të përcaktuar strukturat e kontrollit. Për shembull, të gjitha strukturat standarde të kontrollit në Smalltalk, përfshirë degët (if / else if / else) dhe përsëritësit (for dhe while), përcaktohen duke përdorur objektet, metodat e të cilave pranojnë Closure. Programerët mund të krijojnë strukturat e tyre të kontrollit.

### 3.1.8 Klasat, Instancat, Inicializimi dhe Trashëgimia

Në OOP (Object Oriented Programming) një klasë është një model apo template i zgjerueshëm kodesh për krijimin e objekteve, duke siguruar variabla dhe funksione (metoda) për sjelljen e objektit gjate Runtime-it. Në shumë gjuhë programuese, emri i klasës përdoret si emri për konstruktorin e klasës, dhe si datatipi i objekteve të gjeneruara nga instancimi i klasës. Kur një objekt krijohet nga një konstruktor i klasës, nihet si një instancë e saj, dhe variablat anëtare specifike të objektit quhen variabla të instancës, përndryshe nga variablat e klasës të ndarë për të gjithë klasën. Klasat përbëhen nga pjesë strukturore dhe të sjelljes. Gjuhët programuese që përfshijnë klasat si konstrukte të programimit ofrojnë mbështetje për veçori të ndryshme që lidhen me klasën, dhe sintaksa për të përdorur këto karakteristika ndryshon varësisht nga gjuha programumuese. Përshembull Java ofron aftesinë e krijimit të një objekti statik i cili nuk do të këtë më shumë se një instance përdersia është duke u ekzekutuar, ndërsa ALP për shkak se është gjuhë dinamike e interpretuar dhe e dizajnuar për fillestaret, nuk lejon definimin e një klase apo objekti statik.

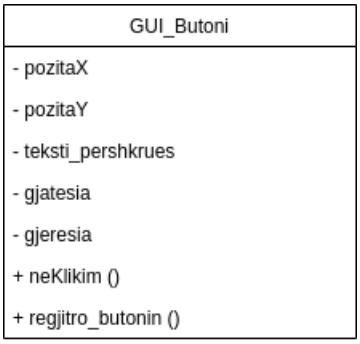


Figura 3.1: UML shema për një klase.

Një klasë përmban përshkrime të fushave të të dhënave (pronat, fushat, anëtarët e të dhënave ose atributet). Këto zakonisht janë fushat dhe emrat që do të egzistojnë si variablat në kohën e ekzekutimit (Runtime) të programit. Këto variabla të gjendjes i përkasin klasës ose instancave specifike të saj. Sjellja e klasës ose e instancave të saj përcaktohet me definimin e metodave. Metoda është grupim i kodit me aftësinë për të vepruar në një objekt. Këto veprime mund të ndryshojnë gjendjen e një objekti ose mundësojnë një menyrë për të komunikuar me ate objekt.

Në kodin e mëposhtëm mund ta shohim një definim të thjeshtë të klasave në ALP. Një aftësi e ALP e cila ngjason me JavaScript është krijimi i variablave në një klase apo objekt në menyrë dinamike, pasi të jetë instancuar. Metoda ”konstruktor” është konstruktori i objektit i

cili mund të jetë pa argumente apo me disa argumente. Nëse nuk ceket një konstruktor, atëherë krijohet një konstruktor i zbrazet në runtime.

**Definimi i klasave dhe instancimi i objektit në ALP**

```
klasa Personi { // klasa nga e cila mund te trashegohet
    konstruktor( emri, mbiemri, mosha ){
        ky.emri = emri;
        ky.mbiemri = mbiemri;
        kjo.mosha = mosha;
    }

    Informata(){
        shkruaj("
            Emri -> " + ky.emri + "
            Mbiemri -> " + ky.mbiemri + "
            Mosha -> " + kjo.mosha
        );
    }
}

// inicializimi i klases se mesiperme ne objekt
deklaro objekti = Personi( "Alixhan", "Basha", Integjer(21) );
objekti.Informata();
```

Tani që kemi kuptuar se cka është një klasë dhe instanca e saj, mund të flasim për trashegiminë në programim. Trashegimia është një koncept i OOP, të cilin e kemi implementuar në ALP. Përmes trashegimisë ne si programerë mund të reprezentojmë marëdhenjet e objekteve në kod. Po ashtu mund të krijojmë një hierarki të kodeve dhe koncepteve programore. Në përgjithësi trashegimia është një menyrë e kodimit e cila mundëson të shkruhet kodi në një formë të renditur ( mbrenda një objekti/klase ) një herë, dhe më pas të mund të ripërdoret pa u rishkruar. Ky edhe është benefiti themelor i trashegimisë. Në kodin e mëposhtem mund ta shohim se si një objekt trashëgon nga një objekt prind. ALP mundëson trashegiminë nga vetem një objekt, ngjajshem me Java-n.

**Trashegimia nga objekti prind në ALP**

```
klasa Studenti trashegon Personi {
    konstruktor( emri, mbiemri, mosha , viti_studimeve){
        super.konstruktor( emri, mosha, viti_studimeve );
        ky.viti_studimeve = viti_studimeve
    }

    Informata(){
        super.Informata();
        shkruaj("
            Viti i Studimeve -> " + ky.viti_studimeve
        );
    }
}
```

.

## 3.2 Implementimi i ALP

Sikurese u përmend edhe me lartë, ALP do të jetë një gjuhë programuese e interpretuar, dinamike dhe me sintaksen në gjuhën Shqipe e implementuar në Java. Në këtë kapitull do të futemi në detaje lidhur me implementimin dhe konceptet e krijimit të një gjuhe programuese. Kapitulli ndahet në dy pjese: Front End dhe Back End. Në një gjuhë si C,C++ apo Java Front end-i përben pjesen e kodit që mund të përdoret pa marë parasysh arkitekturën e procesorit të kompjuterit, pra pjesa e cila tokenizon, analizon dhe rendit kodin e shkruar. Ndërsa Back-End përmban pjesën specifike për arkitekturë, kështuqë kompajlleri i C apo C++ ka Back End specifik për arkitekturat ARM, x86, RISC etj. Kjo do të thotë se në Back End gjindet kodi për gjenerimin e programit të ekzekutueshëm nga procesori, optimizimi i source code-it, kodi i menaxhimit të memorjes dhe në përgjithësi menaxhimi i hardware-it. Pasiqë ALP është implementuar me Java, në përgjithësi Front End-i dhe Back End-i janë të lidhura ngusht me njëratjetrën, gjuha është e interpretuar që do të thotë se nuk ka nevojë për gjenerim të një formati binar të ekzekutueshëm ( executable binary ) pasiqë source code do të ekzekutohet aty-për-aty. Variablat janë dinamike që do të thotë se secila variabël mund të ndryshojë datatipin e vetë, dhe do të veprojë në bazë të rregullave të atij datatipi.

### 3.2.1 Front End

Në këtë pjesë futet ndarja e programit burimor në pjesë përbërëse (tokenizimi) dhe u imponohet një strukturë gramatikore. Më vonë krijohet një përfaqësim i ndërmjetëm (intermediate representation - psh Java Bytecode) të programit burimor. Nëse zbulohet që programi burimor (në aspektin sintaksor dhe semantik) nuk është i qëndrueshëm apo stabil, atëherë duhet të shfaqen mesazhe informuese, në mënyrë që përdoruesi të marrë masa korrigjuese. Pjesa e analizës gjithashtu mbledh informacione rreth programit burimor dhe i ruan ato në një strukturë të dhënash të quajtur një tabelë simboli ( symbol table ), e cila kalohet së bashku me përfaqësimin e ndërmjetëm në pjesën e sintezës.

#### Lekseri dhe Parseri

Hapi i parë në çdo kompajller ose interpretues është skanimi. Skaneri apo lekseri merr kodin burimor të papërpunuar si një seri shkronjash dhe e grupon atë në një seri copash që në i quajmë tokena. Këto janë "fjalët"dhe "pikësimet"kuptimplota që përbëjnë gramatikën e gjuhës sone programuese. në implementimin e ALP me Java, për fazen e parë të perpunimit të kodit burimor janë krijuar klasat Lexer, Token, TokenType dhe Parser. Klasa TokenType është një enum që permban konstantet e simboleve të gjuhes ALP, psh TokenType.PLUS, TokenType.IDENTIFIKUES, TokenType.nëse etj... Klasa Token është një mbeshtjelles i TokenType që permban jo vetem tipin e tokenit të gjeneruar, por edhe informata shtese si vlera e tokenit ( në rrastin e një stringu apo numri të veqante, duhet që vlera e saj të mbahet në një vend në memorje, dhe kjo klase e mundeson këtë opcion ) dhe rreshti ku gjindet ky token. Klasa Lexer lexon kodin burimor të shkruar në ALP dhe mundohet të tokenizoje vlerat që i lexon dhe këto tokena i reprezenton me objektin e klases Token. Shembull mund të marim një deklarim të një variable: 'deklaro n1 = 125;' Lekseri do ta lexoje këtë kod, dhe do ta përkthejë në tokena si në fotografine e mëposhtme

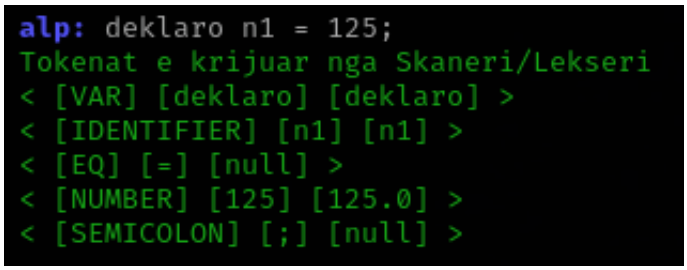


Figura 3.2: Tokenat e gjeneruar nga deklarimi i siperm

Formati i printimit të tokenave është programuar në këtë formë. Fjalët me shkronja të mëdha janë tipet e tokenave të reprezentuara me klasën TokenType, ndrësa pjesa tjetër përmban vlerën si string(tekst) dhe si objekt në memorje. Në këtë pjesë të kodit, lekseri kalon shkronjë pas shkronje deri në fund të kodit, dhe në momentin që mbërin në një hapsirë apo space, shkronjat që i ka lexuar mundohet ti kategorizojë në baze të rreguallve që janë programuar. Këto rregulla janë:

- Nëse shkronjat e skanuara deri tani janë në mes të thojzave, ashtu si ”Hello”, atëherë kemi të bejmë me teks të veqantë, dhe kthejmë një Token të ri në këtë forme: `return new Token( TokenType.TEKST , stringu , stringu , rreshti );`
- Nëse shkronjat e skanuara deri tani janë numra që mund të ndahen me pike ’.’, atëherë kemi të bejmë ose me numra dhjetorë, ose me numra të plotë, dhe kthejmë një Token të ri në këtë forme: `return new Token( TokenType.NUMER , vlera , vlera , rreshti );`
- Nëse shkronjat e skanuara deri tani janë simbole si +,-,*,/,% etj... atëherë kthehet tipi i tokenit në baze të simbolit, dhe kthejmë një Token të ri në këtë formë: `return new Token( TokenType.PLUS , "+", "+", rreshti );`
- Nëse shkronjat e skanuara deri tani janë shkonja të pandara dhe pa thojza, atëherë kemi të bejmë ose me fjalë kyqe të gjuhës, ose me një identifikues ( dmth emër i një variable, funksioni apo klase ) `return new Token( TokenType.IDENTIFIER , vlera_si_string , vlera_si_objekt , rreshti );`

Në figurën e mëposhtme mund ta shohim një pjesë të kodit të klasës Lexer. Kjo është pjesa që përmban logjikën për të përpunuar tokenat të cilat janë pjesë e ALP, por edhe disa simbole që me vonë mund të shtohen. Mund ta shohim se kemi një ’switch’ deklarim i cili e skanon shkronjën të cilën e kemi, dhe në baze të rregullave të cekura përmes deklarimeve ’case’, na kthen një token të caktuar.

```
public void scanToken(){
        char c = next();
        switch( c ){
            case ' ' :
            case '\r':
            case '\t': break;
            case '\n': line++; break;
            case '$': addToken( SUPER ); break;
            case '(': addToken( LEFT_PAREN ); break;
            case ')': addToken( RIGHT_PAREN );break;
            case '{': addToken( LEFT_BRACE ); break;
            case '}': addToken( RIGHT_BRACE );break;
            case ',': addToken( COMMA );      break;
            case '+': addToken( PLUS );       break;
            case '%': addToken( MODULO );     break;
            case '-': addToken( MINUS );      break;
            case '*': addToken( STAR );       break;
            case '.': addToken( DOT );        break;
            case ';': addToken( SEMICOLON );  break;
            case '!': addToken( match('=') ? NOTEQ : BANG );break;
            case '=': addToken( match('=') ? EQEQ : EQ );   break;
            case '<': addToken( match('=') ? LTEQ : LT );   break;
            case '>': addToken( match('=') ? GTEQ : GT );   break;
            // vazhdon
```

Kodi 3.1: Një pjesë e funksionit 'scanToken' nga klasa Lexer

Pasi të kemi tokenizuar kodin burimor me Lekserin e ALP, kalojmë në përpunimin e mëtutjeshëm të gramatikës së gjuhës. Nga aspekti gramatikor, Lekseri vetëm ka shikuar nëse kemi ndonjë simbol të cilin nuk e njohim në ALP, dhe në baze të kësaj na jep error ose na kthen një Token valid. Ndërsa Parseri është faza e ardhshme e "Front End" e cila mer rezultatin e prodhuar nga Lekseri dhe e përpunon edhe më shumë. Po ashtu, në këtë fazë e kemi krijuar klasën "Expression", që do të jetë super-klasa për të reprezentuar shprehjet e ndryshme matematikore dhe programatike si: thirjen e funksioneve, deklarimin e variablave, rregullat për mbledhje, zbritje, shumëzim, pjestim etj... përmes nen-klasave si BinaryExpression, UnaryExpression etj. Duhet të ceket se stili i parserit i cili përdoret në ALP është 'Recursive Descent Parser' dhe është një nga shumë teknologjitë e parserëve. Nuk është më efikse për nga aspekti i memorjes, por është me e thjeshta për implementim dhe poashtu nuk ka nevojë për shikimin e tokenave të kaluar, dhe kjo na mundëson fitim të shpejtesisë. Me Recursive Descent Parser nënkuptojme një parser i cili e skanon kodin burimor nga lartë deri poshtë në menyrë rekursive. Fakti që është rekursive na ndihmon në implementimin e rregullave. Cilat janë këto rregulla ?

- Duhet të respektojë rregullat e matematikës. Në rrastin e një ekuacioni të thjeshtë matematikor si '1 + 2 + 3 * 4' e dijmë që shumëzimi dhe pjestimi kanë përparësi ekzekutimi. Po nëse kemi një ekuacion si '5 * ( 6 + 7 )' . atëherë përparësi kanë numrat mbrenda kllapave. Edhe ALP duhet të veprojë në bazë të këtyre rregullave.
- Rregullat e gjuhës duhet të jenë të kjarta, dhe duhet të përforcohen në këtë fazë. Psh a do të lejojmë që një deklarim i variables të barazohet me një deklarim të një funksioni? A do të suportojmë definime të funksioneve mbrenda deklarimit "nese" apo "if"? A do të lejojmë importimin e një moduli më shumë se një herë? A do të lejojme krijim e variablave të rreja mbrenda blloqeve if? Nëse një variabël e referuar nuk egzsiton si duhet të veprojmë... të kthejmë një error, apo të kthejmë një vlere si ZBRAZET? të gjitha këto pyetje janë valide dhe definimi i tyre e perbën edhe gramatiken dhe semantiken e ALP.

**Shembull kompleks për gramatikën e gjuheve programuese**

```
funksioni numriCift( n ){
    nese( n==0 ) { kthen vertet; }
    kthen numriTek( n-1 ); //thirja e funksionit 'numriTek',
                           // pa ditur nese egziston
}

funksioni numriTek( n ){
    nese( n==0 ) { kthen fals; }
    kthen numriCift( n-1 );
}
```

Në rrastin si më sipër, ku funksioni 'numriCift' mbrenda vetës i referohet funksionit 'numriTek' pa ditur nëse egziston apo jo mund të jetë një problem. Cka duhet të bejmë në një situatë si kjo? Pra Parseri i ALP merret me këto probleme dhe mundohet ti zgjedh ato. Përderisa jemi duke përpunuar kodin me parser, ne jo vetëm që po e kontrollojmë nëse kodi burimor në ALP është i saktë, por edhe po i ndjekim cilat rregulla përputhen me cilat pjesë, në mënyrë që të dimë se cilës pjesë të gjuhës i përket tokeni. Ta konsiderojmë problemin matematikor '6 / 3 - 1'. Dëshirojmë që këtë ekuacion ta kalkulojmë duke përdorur ALP. Parseri ketu mund të veproj në disa mënyra, psh mund që ti jep përparesi pjestimit dhe pastaj zbritjes ( kështu kërkojnë rregullat e matematikës ), ose mund të zbret 3 me 1 dhe pastaj të pjestoje rezultatin me 6, mirëpo metoda e dyte nuk është e sakte! Në të dyja rrastet rezultati është ndryshe.

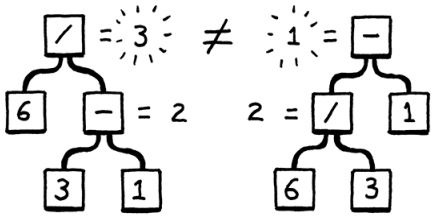


Figura 3.3: Dy metoda të kalkulimit të ekuacionit

Për të zgjedhur këtë problem, duhet të krijojmë rregullat të cilat kanë përparesi ekzekutimi në kod, pra duhet të identifikojmë tokenat dhe në baze të kontekstit ti japim një përparësi. Fatmirësisht ky proces mund të reprezentohet në një formë shumë të kuptueshme dhe po ashtu nuk është edhe shumë e veshtire për implementim. Ketu edhe na hyn në pune rekurzioni. Nëse e shikojmë një program të shkruar në ALP, mund ta shohim se ky program përbëhet nga deklarimet programatike të cilat janë: thirjet e funksioneve me ose pa parametra, deklarimet e variablave, definimet e funksioneve dhe klasave etj. Pra secili rresht i kodit në fakt është një deklarim programatik. Deklarimet përbëhen nga pjesë më të vogla individuale të cilat janë shprehjet programatike ( expressions, janë përmendur në kapitullin e kaluar ). Expressions përbëhen nga pjesë më të vogla e kështu me rradhë. Pra kodi yne për rregullat bazike matematikore dhe të ekzekutimit do të reprezentohet nga funksionet të cilat kanë emrat e rregullave, dhe këto funksione do të lidhen se bashku duke thirur njera-tjetrën me rekurzion. Figura më poshtë e reprezenton këtë proces përmes një flow-diagrami.

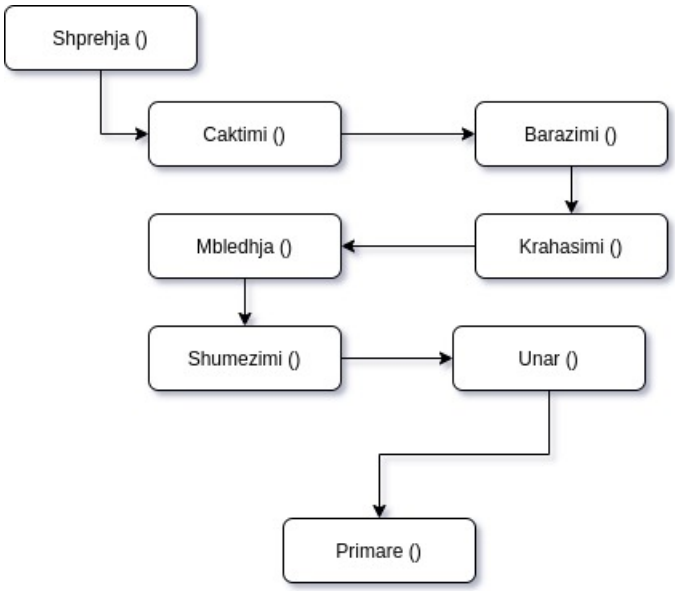


Figura 3.4: Funksionet e parserit që reprezentojne rregullat dhe perparesine e ekzekutimit

Ekzekutimi fillon nga funksioni 'Shprehja', i cili thjesht e thir funksionin 'Caktimi'. Ky funksion pregadit objektet që do të përdoren dhe nëse kushti plotesohet ( gramatika e gjuhës përmban një deklarim të variables, apo ndryshimin e një vlere të një variable ) atëherë kthehet një shprehje caktimi ( assignment expression ). Procesi vazhdon kështu derisa vjen tek rregulli më i vogël të cilin e kupton parseri, dhe ky është një vlerë primitive si një string ose një numër që nuk lidhet me ndonjë rregull tjetër drejtëpërdrejtë. Secili nga këto funksione kthen një Java 'Object' që do të përdoret si reprezentimi i vlerave të ALP, për këtë proces do të flitet më von. Këtu vjen edhe pjesa e gabimeve sintaksore që mund të shfaqen, psh mund të harohet simboli ";" dhe në rrastin e ALP, kjo është një error i sintakses dhe duhet që të raportohet menjëherë në kohën e parsimit. Parseri ka dy detyra themelore:

- Nëse tokenat e krijuar janë valide, prodhon një 'Abstract Syntax Tree' të ekzekutimit
- Nëse hapi i parë është i pasakte, atëherë raporton problemet me kodin.

Një programer nëse nuk ka ide që sintaksa e gjuhes ka gabime, atëherë parseri duhet të raportojë këto probleme dhe ta udhezojë programerin në rrugën e duhur. Mënyra se si bëhet ky lloj i raportimit mvaret nga interface-i i gjuhës programuese. Përballimi i problemeve sintaktike është një problem tejet i rëndë pasiqë kodi burimor është në një gjendeje të definuar me gabime, dhe nuk ka ndonjë mënyrë për të ditur se cfarë qëllimi ka kodi. Sa i përket klasës Expression, kjo klasë përdor "Visitor Pattern" që është një stil i kodit i cili simulon aftësitë e gjuheve programuese funksionale. Arsyeja se përse përdoret ky modeli i programimit është shpejtësia që na ofron. Po të përdornim modelet tradicionale që mundëson Java, do nevoitej përdorimi i 'if' apo 'switch' për të testuar nëse një shprehje e lexuar nga parseri është BinaryExpression, UnaryExpression ose ndonje tip tjeter i Expression, dhe ky proces nuk do të ishte efikas për nga aspekti i shpejtësisë. Ky model programohet duke definuar një interface që përmban metoda të cilat duhen të implementohen nga klasa e cila e trashegon këtë interface. në rrastin e Expression, secili funksion që gjindet në këtë interface si parametër pranon një nën-klase të Expression, dhe ajo nën-klase ka një metodë 'accept' që si parameter pranon një klasë e cila implementon këtë interface. Përmes këtyre hapave largohet edhe nevoja për 'if' ose 'switch' pasiqë operimi i saktë zgjidhet në menyrë automatike duke 'keqpërdorur' një dobesi të Java-s si avantazh.

```
import java.util.List;
public abstract class Expression{
    public abstract <R> R accept( Visitor<R> visitor );
    public interface Visitor<R> {
        public R visitAssignExpression(Assign expression);
        public R visitBinaryExpression(Binary expression);
        public R visitUnaryExpression(Unary expression);
        public R visitLogicalExpression(Logical expression);
        public R visitLiteralExpression(Literal expression);
        public R visitCallExpression(Call expression);
        // ...
    }
    public static class Assign extends Expression {
        public final Token name ;
        public final Expression value;
        public Assign ( Token name , Expression value ) {
            this.name = name;
            this.value = value;
        }
        @Override public <R> R accept( Visitor<R> visitor ){
            return visitor.visitAssignExpression(this);
        }
    }
    // vazhdon
```

Kodi 3.2: Kodi i klases Expression me Visitor Pattern

### Reprezentimi i Kodit

Në ALP, vlerat krijohen nga primitivet ( string ose ndonjë numër ), llogariten nga shprehjet dhe ruhen në variabla. Programeri i sheh këto si objekte të ALP, por këto objekte janë të implementuara në gjuhën programuese me të cilën është shkruar interpretuesi ynë. Kjo do të thotë se datatipet dinamike të ALP janë të lidhura me vlerat statike të Java-s . Një variabël në ALP mund të ruaj një vlerë të çdo lloji (ALP), madje mund të ruajë vlera të llojeve të ndryshme në pika të ndryshme në kohë. Pasiqë jemi duke e krijuar interpretuesin e ALP me Java, a kemi ndonjë datatip që mund ti përmbush kërkesat tona? Duke pasur parasysh një Java variabël me tip statik, ne gjithashtu duhet të jemi në gjendje të përcaktojmë se cilën vlerë mban gjatë kohës së ekzekutimit. Kur interpretuesi ekzekuton një operator binar si mbledhja, duhet të dallojë nëse po mblidhen dy numra, apo po bashkohen dy vargje. Java përmban një datatip që mund ta përdorim si reprezentues të variablave të ALP, dhe ky datatip është java.lang.Object .

| Reprezentimi i vlerave të ALP | |
|---|---|
| ALP Datatipet | Reprezentimi në Java |
| Cfaredo datatipi ALP | java.lang.Object |
| ZBRAZET | null |
| bulean | Boolean |
| numer | Integer ose Double |
| tekst | String |

Tabela e më sipërme tregon datatipet e ALP dhe po ashtu tipin e përdorur për ta implementuar apo reprezentuar në Java. Nëse marim një vlerë të tipit Objekt statik, mund ta determinojmë nëse vlera gjate kohës së ekzekutimit është një numër, tekst apo cfarëdo tipi tjetër përmes një funksioni të Java-s të njohur si ”instaceof”. Duke përdorur aftësine e JVM (Java Virtual Ma-

chine) jemi në gjendje të kuptojmë se cfarë vlere të ALP do të jetë ky Objekt. Kjo metodë mund të përdoret edhe kur dëshirojmë ti reprezentojmë funksionet, klasat, metodat dhe instancat e vetë ALP!

### Vleresimi i Shprehjeve Programatike

Deri në këtë moment me Lekserin dhe Parserin kemi mundësuar që kodi burimor i shënuar në ALP të mund të reprezentohet dhe të kthehet në një Abstract Syntax Tree dhe tani e tutje interpretuesi i ALP do të mer këtë pemë si input. Në këtë faze është krijuar klasa "Interpreter" e cila do të jetë klasa që do ti bashkojë të gjitha pjesët tjera në një, dhe duke marë si input outputin nga klasat e kaluara, do të mundesojë ekzekutimin e kodit të shkruar në ALP. Parase të ekzekutohet kodi duhet që shprehjet (expressions) të mund të vlerësohen, pra kur ne shkruajmë në konsollë apo ndonjë text-editor shprehjen "123 + 321", kjo shprehje duhet të evaluohet dhe të kthehet në rezultatin e duhur duke i ndjekur rregullat e përmendura në faqet e kaluara. Vlerësimin e shprehjeve mund ta bëjmë në forma të ndryshme. Njëra nga këto forma mund të ishte që secilës nyje të pemës sintaksore (shprehjeve) të ia bashkonim një funksion për të interpretuar vete-veten. Kjo metodë përdoret në disa nga gjuhet programuese egzistuese, por në rrastin e ALP për shkak se jemi duke përdorur Java-n si gjuhën për implementim, ideja e krijimit të një klase të veqantë për reprezentimin virtual të interpretuesit është një opsion më i preferuar për nga lexueshmeria e kodit dhe implementimi. Po ashtu kemi edhe "Visitor Pattern" modelin e programimit të spjeguar më sipër, që na ndihmon pikërisht në situatat si kjo. Për të vlerësuar të gjitha shprehjet programatike, duhet që në Java të implemntojmë metoda për secilen strukture që mund të definohet si shprehje programatike, dhe metodat të cilat do të përdoren këtu janë të definuara tek klasa Expression mbrenda interface-it Visitor, dhe pasiqë klasa "Interpreter" implementon Expression.Visitor, atëherë kemi metodat si: visitLiteralExpression, visitGroupingExpression etj... të cilat përmbajne kodin për vlerësimin e strukturave të definuara si shprehje.

### Vleresimi i Deklarimeve Programatike

Përveq klases Interpreter, krijojmë edhe një klasë të ngjajshme me Expression por me emrin Statement, e cila edhe si strukturë ngjan me klasën Expression dhe implementon modelin e vizitorit. Dallimi në mes të këtyre reprezentimeve është se 'Expression' shprehjet japin një rezultat (psh 1+1 është 2) ndërsa 'Statement' deklarimet ndryshojnë ndonjë gjendje të programit ( psh deklarimi i një variable rezulton në rezervimin e një vendi të emërtuar në memorje ). Deklarimet apo 'Statement' përbëhen nga shprehjet apo 'Expression'. Një deklarim mund të jetë krijimi i një funksioni, dhe trupi i ketij funksioni përbëhet prej deklarimeve tjera, të cilat mund të përmbajnë shprehjet që duhet të evaluohen për të vazhduar ekzekutimin. Kjo do të thotë se në rrastin e ALP, deklarimet janë një abstraksion dhe po ashtu një zgjerim i shprehjeve programatike. Për ta kuptuar ALP apo cilendo gjuhë programuese, na mjafton kaq informata lidhur me deklarimet, por në rrastin e ALP deklarimet ndahen në dy grupe: deklarimet të cilat krijojnë një gjendje të re në program dhe deklarimet që modifikojnë një gjendje egzistuese. Të parën e kemi përmendur me siper ( deklarimi i një variable ), ndërsa e dyta është një 'function call' apo thirja e një funksioni. Ky funksion si parametër mund të pranojë vetëm shprehjet Expression, mirëpo mund të këtë rraste ku një vlerë të cilën dëshirojmë ta pasojmë te një funksion (si parameter) gjindet mbrenda një funksioni tjeter, atëherë thirja e një funksioni do të reprezentohet si një hibrid deklarim dhe shprehje. Në më shumë detaje do të flitet në kapitullin **BackEnd -> Funksionet**. Në këtë fazë të interpretuesit është implementuar edhe klasa 'RuntimeError' dhe në përgjithësi arkitektura për raportimin e erroreve që shfaqen në rrastin kur shprehjet apo

deklarimet janë gramatikisht të sakta, por shkaktjonë një gabim të mbrendshëm që do të raportohet gjate kohes së ekzekutimit. Një shembull mund të jetë mbledhja e një numri me një tekst, ky operim në ALP shkakton një RuntimeError i cili raporton gabimin në konsollë dhe më pas ndalon ekzekutimin e interpretuesit, pasiqë kodi i shkruar nuk mund të jep një rezultat të përdorshëm.

### 3.2.2 Back End

Në këtë kapitull do të tregohet procesi i ekzekutimit të kodit burimor. Nëse kapitulli **Front End** ka treguar procesin e ndryshimit të kodit burimor deri në të ashtuquajturën Abstract Syntax Tree, tani e tutje do të tregohet për strukturat e mbrendshme të ALP si 'loop'-at 'conditional operator'-at deklarimet e funksioeve dhe në pergjithesis për rezultatet e shkaktuara nga ekzekutimi i kodit.

#### Ekzekutimi dhe Rezultatet

Kapitulli i kaluar na tregoi se si një tekst file që i përmbahet rregullave gramatikore të ALP kalon nëpër faza të ndryshme të mutacioneve dhe vie deri në piken ku kodi i shkruar arrin në gjendjen e ekzekutueshme. Lekseri, Parseri dhe komponentet tjera bashkëpunojnë për të krijuar një AST dhe kjo strukturë do të mbajë kodin e ALP dhe rezultatin e saj mbrenda Java-s. Në rrastin e ALP FrontEnd dhe BackEnd në fakt nuk janë edhe aq të ndara, pra funksionojnë së bashku dhe në disa rraste mvaren nga njëra-tjetra, mirëpo arsyeja se përse ndahen në teori është për shkak të "portability" apo mbartshmerisë se kodit. Nëse kthehemi në historinë e gjuhëve e vërejmë se kjo mënyrë e ndarjes se implementimit është një taktikë që ndihmon në kohën e krijimit të një gjuhe programuese. FrontEnd përmban kod që nuk mvaret nga arkitektura e kompjuterit, dhe është e njejtë si në x_86 ashtu edhe në arm procesorat, keshtuqë mund të shkruhet një herë dhe të përdoret në të gjitha kompjuteret me arkitetkura të ndryshme, por BackEnd ka pjesën e implementimit që mvaret nga arkitektura, këtu futet analaiza statike e kodit, optimizimi, gjenerimi i ndonjë byte-kodi, garbage collection dhe shumë komponente tjera që kanë neveoje për përdorimin sa më efikas të procesorit në mënyre që rezultati të jetë sa me i shpejtë që është e mundshme. Gjuhët e ndryshme këtu fusin edhe hapa të reja, por në rrastin e ALP edhe BackEnd-i nuk është një pjese jashtëzakonisht e komplikuar. AST e krijuar drejteperdrejtë ekzekutohet nga klasa Interpreter, ndërsa diqka si Java do të krijonte Bytecode të optimizuar. Secila "nyje" e kësaj AST ka kontekstin e vetë, kjo do të thotë se një nyje mund të jetë një deklarim, ndërsa një nyje tjeter mund të jetë një shprehje dhe kjo shprehje mund të jetë një grupim i shprehjeve apo edhe hibrideve deklarim-shprehje të cilat mund të jen shprehje apo deklarime dhe kështu vazhdon cikli derisa file-i përfundon. Secila nyje duhet që në bazë të kontekstit të vetë, të krijojë një rezultat të ditur dhe të parashikuar ashtuqë programeri të jetë në gjendje të krijojë një program stabil duke përdorur rezultatet e parashikueshme. Nëse marim shembull një aftesi bazike të ALP, aftësia e mbledhjes duke përdorur simbolin "+" është një operacion i cili duhet të jetë jo vetëm intuitiv, por edhe i kjartë në ekzekutimin dhe prodhimin e rezultatit. Nëse mbledhim dy numra, rezultati duhet të jetë një numër e jo ndonjë string ose diqka tjeter, pra e dijmë që "+" është një konstante e ALP e cila në varësi të inputit, e jep një output të parashikueshëm. E njejta logjikë aplikohet edhe strukturave të tjera të ALP, pra operatoret dhe strukturat e kushteve, funksionet, klasat etj.

### Kontrollimi i ekzekutimit

Po nëse interpretuesin do ta ndalonim së programuari në pikat e kaluara, atëherë do ta kishim një kalkulator pseudo-primitiv dhe cfarëdo kodi që do ta shënonim, nuk do të kishim kontrolle mbi ekzekutimin e programit. Në gjuhët e vërteta programuese is C, C++, Java, Python egzistojnë blloqet që mundësojnë kontrollimin e ekzekutimit të kodit. Shembull kemi një program që shikon nëse një person ka të drejtë ta merr lejen e vozitjes. Për ta reprezentuar këtë problem na nevoitet që të shtohet struktura ”if-else” (nëse-përndryshe) e kontrollimit. Themi nëse mosha e dhënë është më e madhe se 18, atëherë personi ka të drejte të aplikojë për lejen e vozitjes. Pra përmes këtyre strukturave, jemi në gjendje që të kontrollojmë se cila pjesë e kodit do të ekzekutohet apo jo, dhe kështu mund të krijojmë një program i cili do të reagojë në baze të inputit që ka dhe rezultati mund të jetë ndryshe në varësi se cili grup i kodit është ekzekutuar. Përvec ”if-else” kemi edhe blloqet e përsëritjes si ”for”(për) dhe ”while”(përderisa) të cilat na mundesojnë përseritjen e ekzekutimit të një grupi të kodit përdersia kushti i dhënë është i drejte.

**Shembuj të kontrollimit të ekzekutimit**

```
deklaro mosha = 21;
nese ( mosha >= 18 ){
    shkruaj( "Mund te mer lejen " );
}
perndryshe {
    shkruaj( "Nuk mund ta mer lejen" );
}
// ------------------------------------------ //

per( deklaro i=0; i<10 ; i=i+1 ){ shkruaj( "Hi " + i ); }

// ------------------------------------------ //
deklaro i=0;
perderisa( i<10 ){
    shkruaj( "while " + i );
    i = i+1;
}
```

Mund ta vërejmë në bllokun ”nese” pranohet një kusht, i cili është një shprehje ”expression” dhe më pas përmes kllapave gjarpërore definohet trupi i ketij blloku i cili do të ekzekutohet në rraste se kushti është i vërtetë. Këtu e kemi opsionale definimin e bllokut ”perndryshe”, por kemi edhe një bllok të kushtëzimit e cila është ”tjeter”. Nëse kushtet në blloqet ”nese” dhe ”tjeter” nuk plotësohen, atëherë ekzekutohet blloku ”perndryshe” nëse është i definuar. Eshtë shumë me rëndësi që kodi i shënuar në këto blloqe të shkaterohet në momentin që perfundon ekzekutimi. Bllokun ”per” mund ta shohim se pranon disa gjera, e para është një variabël numer që do të përdoret si numerues, më pas jepet kushti i përsëritjes dmth përderisa kushti është i vërtetë atëherë vazhdo ekzekutimin, dhe e treta është kushti i inkrementimit i cili është një deklarim që tregon për sa do të rritet numeruesi. Sikurse strukturat e mësiperme edhe blloku ”per” ka trupin e vetë që dallohet me kllapat gjarpërore dhe kodi i definuar këtu, duhet të lirohet nga memorja ( shkatërohet ) pasi të përfundon ekzekutimi i trupit. Blloku ”perderisa” është e ngjajshme me atë të ”per” dhe dallimi kryesor është vetëm sintaksa e saj. nN disa rraste është më mirë të përdoret ky bllok mbi ”per”. Një karakteristikë tjetër e ALP është edhe prezenca e operatoreve logjike ”dhe” dhe ”ose” të cilat mundësojnë ekzekutimin e kodit në bazë të rezultatit

të deklarimit apo shprehjes. Në disa rraste kjo strukturë mund të zëvendesojë blloqet ”nese-tjeter-perndryshe” dhe po ashtu konsiderohet të ekzekutohet më shpejtë, pasiqë jo vetem që ka më pakë sintakse por edhe në praktike kontrollon qarqet elektrike të kompjuterit drejtë-përdrejtë duke shkaktuar një ”short-circuit” jo dëmtues, i cili parandalon ekzekutimin e kodit të padëshiruar.

**Kontrolli i ekzekutimit përmes operatoreve logjik**

```
funksioni test( d ){
    nese( d==1 ){ kthen vertet; }
    kthen fals;
}

test(1) dhe shkruajr("OK"); // ekzekutohet
test(2) dhe shkruajr("OK"); // nuk ekzekutohet

test(1) ose shkruajr("OK"); // nuk ekzekutohet
test(2) ose shkruajr("OK"); // ekzekutohet
```

### Funksionet

Funksionet në cfarëdo gjuhë programuese mund të përshkruhen si blloqe të kodeve të cilat kryejnë një detyrë të caktuar, me kodin e shkruar nga programeri. Edhe ALP e ndjek këtë filozofi. Një funksion në ALP është një definim i një objektivi të caktuar, psh funksioni ”shkruaj” do të printojë në ekran vlerën e dhënë. Kjo vlere mund të jetë një numër, tekst, shprehje apo edhe deklarim i cili kthen një shprehje të caktuar. Në këtë fazë të implementimit janë krijuar klasat: ALPCallable, ALPNativeInterface, ALPFunction, Return dhe shumë klasa tjera të cilat mundesojnë kuptimin dhe ekzekutimin e funksioneve të definuara nga programeri. Funksionet në ALP egzistojne për arsyen e definimit të një operimi të caktuar. Duke përdorur aspektet tjera të gjuhes programuese, mund të krijojmë apo definojmë një set të rregullave që do të japin një rezultat të parashikueshëm. Një funksion dallohet nga strukturat tjera të gjuhes prej sintaksës së vetë. Kjo sintakse përbëhet nga fjala kyqe ”funksioni” , emri i këtij funksioni i cili mund të jetë cfarëdo, kllapat e vogla që do të mbajnë argumentet ( variablat të cilat mund ti jepen funksionit nga një pjese tjeter e kodit ) dhe më në fund kllapat gjarperore që reprezentojnë fillimin dhe fundin e trupit të funksionit. Kodi i mëposhtëm përshkruan sintaksen e funskioneve në ALP.

**Llojet e definimit të një funksioni në ALP**

```
funksioni bosh(){} // nje funksion qe nuk ben asgje
funksioni test1(){ shkruaj( "Funksioni i cili punon" ); }
funksioni test2( argumenti ){
    shkruaj( "Funksioni i cili pranon nje argument -> " + argumenti );
}
// sintaksa per thirjen e funksioneve te definuara
bosh();
test1();
test2( "arg" );
```

Përveq se që mund të definojmë një funksion, ne po ashtu mund ta ”thirrim” atë. Kjo do të thotë se funksionin e definuar mund ta ekzekutojme përmes sintakses: emri i funksionit dhe me pas kllapat e vogla dhe në fund pike-presje. Varesisht nga lloji i funksionit, ky mund të pranojë argumente dhe këto argumente mund të jenë vlera direkte ( 1, 5.3, "tekst") apo variabla egzistuese që kanë ndonje vlere. Përveq kësaj, një funksion si parameter mund të pranojë edhe një thirje të një funksioni. Në këtë rrast, funksioni i cili thiret si parameter duhet të kthejë një vlere, ashtuqë thirja e funksionit në kohen e ekzekutimit të zëvendesohet me vlerën që e kthen ky funksion. Shembull: **test( 1, tjeter() );** <- në këtë kod mund të vërejmë se një thirje e një funksioni është pasuar si parametër, në kohën e ekzekutimit ky funksioni ekzekutohet më përpara dhe vlera e kthyer zëvendësohet me thirjen, ashtuqë kodi i mësipërm zevendesohet me: **test( 1, "vlera e funksionit");** . Një aftësi tjeter e funksioneve në ALP është edhe modeli i ”Closures”-ave, koncept për të cilin kemi folur në kapitujt e kaluar. Një funksion mund të mbajë disa funksione tjera mbrenda vetes, dhe kështu është në gjendje të imitojë një objekt primitiv i cili mund të këtë vlera private ( të pa aksesueshme ) mbrenda vetës. Eshtë më rëndesi të ceket se mënyra e implementimit të funksioneve të ALP bazohet pikerisht në kllapat e vogla për thirjen e këtij funksioni. Funksionet në prapavijë ruhen në të njëjtin vend si variablat, dhe ruhen si variabla që kanë një vlerë të caktuar dhe në këtë rrast vlera është një bllok i kodit që kryen një detyrë të caktuar. Si efekti i kësaj mënyre të implementimit, nëse mundohemi të thirim një variabel si funksion, nuk do të shfaqet një error i sintaksës, por një error i runtime-it i cili na thot që vetem funksionet apo metodat mund të thiren. Duke e ditur se iterpretuesi e din që një thirje e funksionit bazohet në kllapat e vogla, në definimin e një funksioni mund të krijojmë një parametër i cili automatikisht ka vleren ”ZBRAZET”, mirëpo kur e thirim atë funksion për tu ekzekutuar, ia pasojmë një funksion tjeter jo si thirje por si emër dhe pasiqë variablat në ALP janë dinamike, një variabel mund të kthehet edhe në funksion, dhe me pas të thiret për ekzekutim. Ky proces nihet si callback, dhe është një koncept i krijuar nga gjuhët programuese funksionale por edhe gjuhët si Python dhe JavaScript kanë implementuar këtë teknologji.

**Demonstrimi i Callback-ave**

```
funksioni callback(){
    shkruajr("Hello World");
}

funskioni test( nje_funksion ){
    nje_funksion ();
}

test( callback );
```

Në anën e Java-s rëndësi të madhe në implementim ka interface-i ”ALPCallable”. Ky është një interface që ka dy funksione dhe qellimi i këtij interface-i është që të përdoret si në funksionet, ashtu edhe në metodat e klasave, dhe përdoret si reprezentues i një objekti që mund të ”thiret” apo ekzekutohet. Nëse një klasë implementon këtë interface, atëherë automatikisht bëhet pjesë e ekzekutueshme e ALP. Një klasë e cila e implementon këtë interface është klasa ”ALPFunction” e cila përmban metoda ndihmesë falë të cilave mund të mblidhen informata për funksionet e definuara në ALP dhe të ekzekutohen ato. Këtu përdoret edhe klasa ”Return” e cila mundëson kthimin nga një funksion apo metode e definuar në ALP. Duhet të ceket se duke përdorur ALPNativeInterface janë shtuar funksionet e ”standard library”. Ky interface është një urë lidhese në mes të ALP dhe Java-s. Interface-i pranon kod të Java-s dhe në runtime e përkthen

ate në ALP kod që mund të përdoret në programet e krijuara me ALP. Disa nga funksionet e definuara në ALPNativeInterface janë: Input, instanca_e, Integjer, String, Dhjetor dhe disa funksione që mundësojnë kontrollimin e variablave dhe tipeve të tyre. Këtu janë shtuar edhe funksionet për input/output në file-a si FLEXO dhe FSHKRUAJ, metoda këto që mundësojnë leximin e një file-i dhe shkrimin në një file respektivisht.

```
// Definon nje ALP funksion nativ
// i cili eshte i implementuar ne java,
// por punon ne ALP skriptat e shkruara.

environment.define( "Integjer" , new ALPCallable(){
    @Override public int ArgumentSize(){ return 1; }
    @Override public Object FunctionCall( Interpreter i, List<Object>args )
    {
        int ret = 0;
        try {
            ret = (int)Double.parseDouble((String)args.get(0).toString());
        } catch(NumberFormatException nfe){
            throw new
                RuntimeError( "--[ "+args.get(0).toString() +
                    " ]-- nuk mund te konvertohet ne Integjer"
                );
        }
        return ret;
    }
    @Override public String toString()
    {
        return "<Funksion nativ => 'Integjer'>";
    }
});

// vazhdon
```

Kodi 3.3: Kodi nga ALPNativeInterface

### Klasat

Deri në këtë moment mund të themi se kemi përshkruar dhe implementuar krijim e një gjuhe programuese të vërtetë. ALP është në gjendje të kontrollojë memorjen duke ndarë vend për krijimin e variablave, mund ta kuptojë konceptin e fusheveprimeve të ndryshme, është në gjendje të definojë dhe ekzekutojë edhe funksionet. Nëse do të ishim në vitet e 80-ta, atëherë ALP do ta thirnim si një gjuhë të kompletuar programuese e cila ngjason me C nga ana e sintakses. Mirëpo me klasat ne shkojmë një hap më përpara, dhe me këtë hap na shfaqen edhe probleme tjera si: Sintaksa e re për reprezentim të klasave, instacimi i tyre, cka nëse dëshirojmë të shtojmë trashegiminë e objekteve? Si do të bashkëpunojë ky koncept i ri me kodin që egziston? Klasat ( dhe objektet si efekti anësor ) mund ti mendojmë si super-funksione që janë në gjendje të kontrollojnë jo vetem disa funksione por edhe vlera të caktuara si variablat e objektit. Edhe implementimi i klasave ngjan me ate të funksioneve. Në këtë hap të interpretuesit janë definuar klasat: ALPClass dhe ALPInstance. ALPClass implementon interfacein e krijuar në hapin e mëparëm ALPCallable e cila definon funksionet të cilat mundësojnë ekzekutimin e funksioneve, dhe sikur e cekëm më siper klasat janë super-funksione dhe nëse dëshirojmë ti ekzekutojmë metodat e definuara mbrenda klasve, ky hap është i nevojshëm. Për arsye të thjeshtësisë secila klasë me ”fëmijët” e saj ( variablat dhe metodat ) janë publike dhe nuk ka instanca statike. Klasa ALPClass implementon jo vetëm metoda për të konsumuar parametrat e

dhëna nga konstruktori, por përmban edhe metoda për të shikuar nëse një metode apo variabël egziston në atë klasë. Kështu nëse programeri mundohet të manipulojë vlerat jo-egzistuese, atëherë interpretuesi do të paralajmerojë këtë qështje. Diqka e ngjajshme me JavaScript-in është që pasi të krijohet dhe instancohet klasa dhe objekti i saj, objektit mund ti shtohen vlera të reja, por kjo ndodh me vetë-dijeninë e programerit. Ashtuqë nëse një objekt "Personi" nuk e ka të definuar variablen "emri", pasi të jete instancuar objekti mund të bëhet definimi në menyrë dinamike. Kjo aftësi mund të përdoret edhe me funksionet. Sintaksa e klases në ALP përbëhet prej fjales-kyqe "klasa", më pas emri i klasës dhe kllapat gjarpërore që reprezentojnë trupin e kësaj klase. Një klasë mund të ketë një konstruktor që është opsionale, pra nëse konstruktori nuk definohet atëherë klasa do ta krojojë një konstruktor bosh që perdoret për inicializim të objektit. Shkaterimi i një objekti bëhet në menyre automatike nga JavaGarbageCollector në momentin që ky objekt nuk përdoret në kodin burimor të ALP. Metodat të cilat janë pjese e një objekti apo klase, në fakt janë funksione dhe dallimet janë se kur një metode definohet mbrenda një klase, nuk ka nevojë për ndonjë fjale-kyqe. Po ashtu metodat e definuara në një klasë janë pjese e asaj klase dhe mund të ekzekutohen apo thirren në momentin që klasa është instancuar apo inicializuar.

**Sintaksa e klasave dhe inicializimi**

```
klasa Personi { // definimi i klases
    konstruktori( emri, mbiemri ){
        ky.emri = emri;
        ky.mbiemri = mbiemri;
    }
    Info(){
        shkruajr( "Emri " + ky.emri );
        skhruajr( "Mbiemri " + ky.mbiemri );
    }
}

deklaro p = Personi( "Alixhan" , "Basha" ); // inicializimi i objektit
p.Info(); // afishon ne ekran emrin dhe mbiemrin e onjektit
```

Ka disa arsye se përse klasat egzistojnë. Arsyeja e parë përse klasat egzistojnë është se në një gjuhë programuese duhet që të mbajë të dhenë ( data ) lidhur me një objekt që do të krijohet në kohen e ekzekutimit, ndërsa arsyeja e dytë është se kodi i shkruar mund të ripërdoret pa u riimplementuar dhe pikesisht këtu futet edhe trashëgimia në një gjuhë programuese të orientuar në objekte si ALP. Në këtë rrast trashëgiminë mund ta definojmë si një koncept që mundëson përdorimin e vlerave dhe metodave të një super-klase, nga një klasë dytësore që i "trashegon" ato. Ashtuqë nëse kemi një super-klasë Personi që përshkruan një person, kjo në fakt është një template që përdoret nga një nën-klasë Studenti, apo Punetori dhe kodi për definimin e personit nuk duhet të ri-implementohet në këto nën-klasa. Në fakt nën-klasa Studenti mund të definojë një person që është student, pra futet në detaje. Në jeten reale ky lloj i programimit mund të përdoret për reprezentimin e koncepteve të gjera, dhe me pas duke perdorur këto "template"-a mund të specifikohet subjekti i dëshiruar. Një klasë mund ti këtë disa instanca, ashtuqë klasa Personi mund ti ketë ndoshta dhjetë instanca në një program. Në menyre që të reprezentohen vlerat të cilat janë pjese e klases dhe me vone edhe të instances në këtë faze të interpetuesit u shtua suportim për fjalet kyqe "ky" dhe "kjo". Këto pjese të gjuhes u referohen instances aktive që është në perdorim e siper. Psh kemi kodin e mëposhtëm:

**Instancat dhe qasja e tyre**

```
klasa Personi {
    konstruktori( emri, mbiemri ){
        ky.emri = emri;
        ky.mbiemri = mbiemri;
    }
    Info(){
        shkruajr( "Emri " + ky.emri );
        skhruajr( "Mbiemri " + ky.mbiemri );
    }
}

deklaro p1 = Personi( "Alixhan" , "Basha" );
deklaro p2 = Personi( "Filan" , "Fisteku" );
deklaro p3 = Personi( "Student" , "Studenti" );
p1.Info();
p2.Info();
p3.Info();
```

Ky kod demonstron se mund të kemi disa instanca të të njejtës klasë dhe nëse dëshirojmë të përdorim metodat dhe variablat e definuara në atë instancë duhet të përdorim këtë sintakse. Definimi i klases përmban fjalen-kyqe ”ky”, dhe kjo mundëson që në runtime objekti i krijuar nga ky template të mund ti referohet vetes. Në rrastin e ketij kodi, variabla ”p3” është instace e klases Personi dhe fjala-kyqe ”ky” referohet variables apo objektit ”p3”. Në rrastet ku perdoret ”ky” dhe ”kjo” u referohemi instancës se klasës, por cka nëse dëshirojmë të referohemi një super-klase. Në ate rrast përdorim fjalen-kyqe ”super”. Kjo na jep qasje në super-klasen nga e cila klasa jonë trashegon. Nëse përdoret pa pasur ndonjë super-klasë atëherë raporton një error.

**Super-klasa dhe Nen-klasa**

```
klasa Personi {
    konstruktori( emri, mbiemri ){
        ky.emri = emri;
        ky.mbiemri = mbiemri;
    }
    Info(){
        shkruajr( "Emri " + ky.emri );
        skhruajr( "Mbiemri " + ky.mbiemri );
    }
}
klasa Studenti trashegon Personi {
    konstruktori( emri, mbiemri, viti_studimeve ){
        super.kosntruktori( emri, mbiemri );
        ky.viti_studimeve = viti_studimeve
    }
    info(){
        super.info();
        shkruajr( "Viti i studimeve: " + ky.viti_studimeve );
    }
}
```

# Kapitulli 4

# Testimet, Krahasimet dhe Konkluzioni

Ky kapitull përmban testimet të cilat i kam kryer në sistemin tim. Dallimi në mes versioneve të Java-s dhe poashtu mendimet e mia lidhur me këtë punim diplome, ku mund të aplikohet dhe cilat janë aspektet pozitive dhe negative të këtij implementimi të ALP dhe në fund, si mund të përmisohet.

### 4.0.1 Krahasimi i JDK14, JDK11 dhe Native Binary

Sikur u përmend disa herë në tekst, ALP është implementuar me Java. Verzioni i përdorur ka qenë ”open-jdk-14”. Në përgjithësi JVM është në gjendje që të optimizojë kodin e shkruar për të rritur shpejtësinë e ekzekutimit të programit. Versionet e ndryshme të Java-s kanë menyra të ndryshme të optimizimit dhe si pasojë e kesaj, kam kompajlluar kodin e shkruar në tre forma të ndryshme.

- Java open-jdk-14: në përgjithesi versioni me stabil dhe poashtu optimizimi gjatë runtime-it i aplikuar nga ky verzioni i JVM ishte më i suksesshmi. Optimizimi i kodit vërehet më së shumti gjatë iterimit me blloqet ”per”, ”perderisa” dhe rekurzionit të funksioneve të ALP. Koha mesatare e ekzekutimit të skriptes të shkruar në ALP, për të testuar shpejtesine është rreth 5 sekonda. Skripta përmban definim dhe ekzekutim të funksioneve, variablave, blloqeve përseritëse dhe një funksion i cili kalkulon numrat fibinaqi deri në 30. Ky i fundit është një test i cili përdor rekursionin dhe si pasojë mer më së shumti kohë.
- Java open-jdk-11: Për nga aspekti i performancës është me i ngadalshmi. Optimizimi i kodit në këtë verzion të JVM nuk është në nivelin e open-jdk-14, dhe si pasojë shpejtesia e fituar nga optimizimi i kodit në rekurzion dhe iterim është me i ulet. Poashtu shkrimi i tekstit në konsole, ekzekutimi i funksioneve dhe qasja në variabla krahas me open-jdk-14 është më e ngadalshme. Koha mesatare e ekzekutimit të ALP skriptes për testim është 7-8 sekonda.
- GraalVM jdk-11 / Native Binary: Kjo teknologji është ende e re, dhe [10]GraalVm është një makinë viruale sikur JVM e cila mundëson një runtime më të shpejtë për disa gjuhë programuese. Një aftësi e kësaj teknologjie është që të prodhojë ”native binary executable” për sistemin operativ dhe arkitekturën e procesorit të kompjuterit. Kjo do të thotë se kodi i shkruar në Java, kthehet në formatin binar të sistemit, e jo në Java Bytecode. Aspektet pozitive të kësaj teknologjie janë se formati binar i gjeneruar është më i shpejtë se ai i Bytecode-it, por optimizimi i aplikuar nga JVM nuk transmetohet në këtë

format. Si pasojë e kesaj prapambeturie, rekurzioni dhe iterimi nuk kanë optimizim. Por ekzekutimi i kodit, pra tokenizimi, parsimi, ekzekutimi, shkrimi në konsollë në përgjithësi janë më të shpejta se dy verzionet e sipër-përmendura. Koha mesatare e ekzekutimit të ALP skriptës për testim është 8-9 sekonda.


Figura 4.1: Grafi i cili tregon dallimin e shpejtesise se ekzekutimit ( me ulet, me mire )

### 4.0.2 Konkluzioni

Një ide për aplikimin e këtij projekti në jetën reale do të ishte krijimi i një platforme dhe materialeve edukative për fillestaret e programimit. Edhepse gjuha është e dizajnuar që të jetë e lehtë për cfarëdo gupmoshe, vizioni im për këtë projekt do të ishte që ALP të mësohej në shkolla fillore apo të mesme profesionale si një alternativë për gjuhët tjera programuese. Arsyeja se përse besoj që do të kishte sukses është në fakt sintaksa, e cila është në gjuhën Shqipe. Kështu individet të cilët dëshirojnë të mësojnë programimin, mund ta bëjnë në gjuhën amëtare pa pasur nevojë të mesojnë gjuhën Angleze.

**Aspektet pozitive të ALP:**

- E lehte për fillestarë të grupmoshave të ndryshme
- E dizajnuar në gjuhën Shqipe
- E ngjajshme me JavaScript në disa aspekte, që do të thotë se mund të bëhet tranzicioni me i lehtë.
- Mund të shpreh probleme komplekse duke pasur parasysh se i gjith kodi burimor për interpretuesin e ALP është 2590 rreshta të Java-s.

- Mund të përdoret si ”glue language”.
- OOP është e kapshme dhe jokomplekse

**Aspektet negative të ALP:**

- E ngadalshme pasiqë është e implementuar në Java.
- Nuk ka reprezentim të vargjeve
- Libraria standarde është e limituar

**E ardhmja e ALP**

Që ALP të jetë një gjuhë e cila konkuron me gjuhët si Python, Java dhe të tjerat ka nevojë ende për përmisime. Ndryshimet që do ti kisha implementuar me diturine që kam fituar me krijim e kësaj gjuhe programuese është se fillimish do ta përdorja një gjuhë si C, C++ apo edhe ndoshta Rust. Këto janë gjuhë të niveleve më të uleta, dhe si pasojë kanë kontroll më të lartë të hardware-it, nuk kanë ”Runtime Overhead” ( ngadalsime përgjat kohës së ekzekutimit ) si garbage collector, konvertim të variablave në menyrë dinamike etj. Do të kisha shtuar mënyra për të reprezentuar disa vlera në të njejtën kohë ( arrays apo vargjet ) dhe në fund, do të mundohesha të krijoj një librari standarde të kodit që miret me input/output, socket-a për rrjete, multi-thread aftësi dhe në përgjithësi aftësitë që një gjuhë programuese pritet ti posedojë.

# Bibliografia


[1] John Fuegi **and** Jo Francis. **?**Lovelace & Babbage and the creation of the 1843'notes'**?** **in**: *IEEE Annals of the History of Computing* 25.4 (2003), **pages** 16–26. (kontrolluar: 06/2020) (**backrefpage** 3).

[2] Steven Colyer. *The NAND Gate - One Gate to Rule Them All*. 2011. URL: https://tetrahedral.blogspot.com/2011/02/nand-gate-one-gate-to-rule-them-all.html. (kontrolluar: 11/2020) (**backrefpage** 5).

[3] Computer Hope. *What is Machine Language*. 2019. URL: https://www.computerhope.com/jargon/m/machlang.htm. (kontrolluar: 09/2020) (**backrefpage** 5).

[4] John H. Reynolds. *Bootstrapping a self-compiling compiler from machine X to machine Y*. 2003. URL: https://dl.acm.org/doi/10.5555/948785.948811. (kontrolluar: 10/2020) (**backrefpage** 6).

[5] Digital Guide. *What is Source Code*. 2020. URL: https://www.ionos.com/digitalguide/websites/web-development/source-code-explained-definition-examples/. (kontrolluar: 11/2020) (**backrefpage** 7).

[6] Paola Andrea Noreña Cardona. **?**Compilers: Principles, Techniques, and Tools**?** **in**: (). (kontrolluar: 05/2020) (**backrefpage** 8).

[7] Alfred V Aho, Monica S Lam, Ravi Sethi **and** Jeffrey D Ullman. *Compilers: Principles, techniques, and tools second edition*. 2007. (kontrolluar: 05/2020) (**backrefpage** 8).

[8] www.cs.man.ac.uk. *Anatomy of a Compiler*. URL: http://www.cs.man.ac.uk/~pjj/farrell/comp3.html. (kontrolluar: 06/2020) (**backrefpage** 9).

[9] Denis Howe. *Free on-line Dictionary of Computing (FOLDOC)*. 1996. (kontrolluar: 09/2020) (**backrefpage** 10).

[10] GraalVM Team. *GraalVM Documentation*. 2020. URL: https://www.graalvm.org/docs/introduction/. (kontrolluar: 01/2021) (**backrefpage** 34).