УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА

УНИВЕРЗИТЕТ У НОВОМ САДУ
ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА
НОВИ САД
Департман за рачунарство и аутоматику
Одсек за рачунарску технику и рачунарске комуникације

ПРЕДМЕТНИ ПРОЈЕКАТ

Кандидат: **Кристина Младеновић**
Број индекса: **РА 242-2022**

Предмет: **Системска програмска подршка у реалном времену I**
Тема рада: **МАВН - ПРЕВОДИЛАЦ**

Ментор рада: **др Миодраг Ђукић**

Нови Сад, јун, 2024.

Sadržaj

1. UVOD

1.1 Problem i cilj projekta

U ovom delu će se identifikovati osnovni problemi koje projekat ima za cilj da reši. Biće detaljno razmatrani izazovi koji se javljaju pri prevodu programa sa višeg assemblerskog jezika na osnovni MIPS 32-bit assemblerski jezik. Pored toga, biće jasno definisani ciljevi projekta, odnosno šta se očekuje da projekat postigne u rešavanju ovih problema.

1.2 Značaj i upotreba prevodioca

Ovaj segment će istražiti značaj prevodioca u domenu razvoja softvera. Biće naglašena važnost pravilnog i efikasnog prevoda programa kako bi se omogućilo izvršavanje na određenoj platformi. Takođe će se istaći različite primene prevodioca u realnim scenarijima, kao i kako prevodioci doprinose procesu razvoja softvera.

2. ANALIZA PROBLEMA

2.1 Višeg assemblerski jezik

Viši assemblerski jezik predstavlja složeniju sintaksu i semantiku u odnosu na osnovni assemblerski jezik. U ovom delu će se analizirati karakteristike višeg assemblerskog jezika, kao i specifičnosti koje mogu predstavljati izazov prilikom njegovog prevoda na osnovni MIPS 32-bit assemblerski jezik. Razumevanje višeg assemblerskog jezika ključno je za razvoj efikasnog prevodioca koji može pravilno tumačiti i generisati odgovarajući kod.

2.2 Osnovni MIPS 32bit assemblerski jezik

Osnovni MIPS 32-bit assemblerski jezik predstavlja ciljni jezik na koji će se prevoditi programi iz višeg assemblerskog jezika. U ovoj sekciji će se detaljno analizirati karakteristike osnovnog MIPS 32-bit assemblerskog jezika, uključujući registre, instrukcije i specifičnosti arhitekture. Razumevanje ovih karakteristika od suštinskog je značaja za uspešno prevodjenje programa i generisanje ispravnog koda koji se može izvršiti na MIPS 32-bit arhitekturi

2.3 Zahtevi za prevodioca

Ovde će se razmotriti osnovni zahtevi koje prevodilac treba da ispuni kako bi bio efikasan i pouzdan. To uključuje detekciju i obradu leksičkih i sintaksičkih grešaka, kao i generisanje odgovarajućih izveštaja o greškama. Prevodilac takođe treba da bude u stanju da generiše ispravan assemblerski kod koji se može izvršavati na MIPS 32-bit arhitekturi, uzimajući u obzir sve specifičnosti i ograničenja ovog jezika.

3. KONCEPT REŠENJA

3.1 Arhitektura prevodioca

Arhitektura prevodioca predstavlja strukturu i organizaciju softverskog sistema koji će obavljati prevodjenje programa sa višeg assemblerskog jezika na osnovni MIPS 32-bit assemblerski jezik. Ovaj deo će detaljno razmotriti arhitekturu prevodioca, uključujući komponente, slojeve i njihove međusobne veze. Ključni elementi arhitekture biće identifikovani i opisani kako bi se omogućilo razumevanje celokupnog sistema prevodioca.

3.2 Faze prevodjenja

Faze prevodjenja predstavljaju korake kroz koje prolazi prevodilac prilikom obrade izvornog koda programa. Ovaj deo će razmotriti osnovne faze prevodjenja, uključujući leksičku analizu, sintaksnu analizu i generaciju koda. Svaka faza će biti detaljno opisana, zajedno sa svojim ulazima, izlazima i ključnim zadacima koje obavlja. Razumevanje faza prevodjenja od suštinskog je značaja za razvoj efikasnog prevodioca.

3.2.1 Leksička analiza

Leksička analiza je prva faza prevodjenja koja se bavi analizom izvornog koda programa i njegovog razlaganja na leksičke jedinice, poput tokena. Ovaj segment će detaljno opisati proces leksičke analize, uključujući rad leksičkog analizatora, pravila tokenizacije i prepoznavanja leksičkih elemenata. Takođe će se istražiti kako leksička analiza doprinosi daljem procesu prevodjenja programa.

3.2.2 Sintaksna analiza

Sintaksna analiza je faza prevodjenja koja se bavi strukturom izvornog koda programa i proverom da li se program pravilno uklapa u sintaksu jezika. Ovaj deo će istražiti proces sintaksne analize, uključujući rad parsera, pravila gramatike i sintaksne strukture jezika. Biće analizirani različiti pristupi sintaksnoj analizi, kao i njihova primena u razvoju prevodioca.

3.2.3 Generacija koda

Generacija koda je poslednja faza prevodjenja koja se bavi pretvaranjem sintaksno ispravnog izvornog koda programa u odgovarajući ciljni kod, u ovom slučaju MIPS 32-bit assemblerski jezik. Ovaj segment će istražiti proces generacije koda, uključujući rad generatora

koda, pravila generisanja instrukcija i optimizaciju generisanog koda. Biće analizirani različiti pristupi generaciji koda i njihova primena u praksi.

4. PROGRAMSKO REŠENJE

4.1 Constants.h

Constants.h je zaglavlje koje sadrži definicije konstanti koje se koriste u programu. Ove konstante obuhvataju simbole stanja, broj stanja u konačnom stanju automatu (FSM), broj podržanih karaktera, oznake za interferenciju i prazninu između instrukcija, broj registara u procesoru, kao i oznake za ispisivanje analize života. Takođe, sadrži definicije za poravnanje radi lepšeg ispisivanja. Ovaj fajl pruža jednostavan način da se konstante koriste u programu bez potrebe za ponovnim definisanjem.

4.2 Types.h

Types.h je zaglavlje koje sadrži definicije tipova podataka i enumeracija koje se koriste u programu. Ove definicije obuhvataju tipove tokena, tipove instrukcija, imena registara i druge relevantne tipove. Korišćenje ovih definicija omogućava jednostavno i čitljivo korišćenje različitih tipova podataka u programu, što olakšava razvoj i održavanje koda. Ovo zaglavlje omogućava konzistentno korišćenje tipova širom programa bez potrebe za ponovnim definisanjem.

4.3 Token.h

Token.h je zaglavlje koje definiše klasu Token koja predstavlja token u programu. Token predstavlja osnovnu jedinicu u procesu leksičke analize, koja se sastoji od tipa i vrednosti. Ovo zaglavlje takođe sadrži deklaracije funkcija za rad sa tokenima, uključujući funkcije za postavljanje i dohvatanje tipa i vrednosti tokena, kreiranje tokena na osnovu ulaznih podataka, kao i funkcije za ispisivanje informacija o tokenima. Korišćenje ove klase omogućava lakše rukovanje tokenima u procesu leksičke analize programa.

4.3.1 Token.cpp

Token.cpp je datoteka koja implementira metode klase Token definisane u zaglavlju *Token.h*. Ove metode obavljaju različite funkcije vezane za rad sa tokenima, kao što su postavljanje i dohvatanje tipa i vrednosti tokena, kreiranje tokena na osnovu ulaznih podataka, generisanje tokena greške ili tokena za kraj datoteke, kao i ispisivanje informacija o tokenima. Takođe, u ovoj datoteci se nalazi i funkcija tokenTypeToString koja prevodi tip tokena u odgovarajući string koji se koristi prilikom ispisivanja informacija o tokenima. Ova datoteka igra ključnu ulogu u procesu leksičke analize programa, omogućavajući identifikaciju i obradu tokena u skladu sa definisanim pravilima jezika.

4.4 IR.h

IR.h je zaglavlje koje definira klase i tipove koji se koriste za reprezentaciju unutrašnje reprezentacije (IR) programskog koda. Ovo zaglavlje sadrži definicije klasa Variable i Instruction, kao i tipova Variables i Instructions.

Klasa Variable predstavlja jednu promenljivu iz programskog koda i sadrži informacije o njenom tipu, imenu, poziciji i dodeli registra (ukoliko je promenljiva registarskog tipa). Ova klasa takođe omogućava manipulaciju promenljivama, kao što su dohvatanje imena, tipa i vrednosti, kao i postavljanje dodele registra.

Klasa Instruction predstavlja jednu instrukciju u programskom kodu i sadrži informacije o njenom tipu, poziciji, odredištima, izvorima, upotrebi, definiciji i susednim instrukcijama. Ova klasa omogućava dodavanje oznake instrukciji, dodavanje odredišta i izvora, postavljanje upotrebe i definicije, kao i dohvatanje informacija o instrukciji.

Osim toga, u ovom zaglavlju se nalaze i pomoćne funkcije i tipovi koji se koriste u radu sa unutrašnjom reprezentacijom programa.

4.4.1 IR.cpp

IR.cpp je datoteka implementacije koja sadrži definicije metoda članica klasa Variable i Instruction koje su deklarisane u zaglavlju *IR.h*. Ova datoteka takođe sadrži implementaciju pomoćnih funkcija koje se koriste u radu sa promenljivama i instrukcijama.

U datoteci *IR.cpp* definišu se metode članice klase Variable, kao što su metode za dohvatanje imena, tipa, vrednosti i dodele promenljive, kao i metode za ispisivanje informacija o promenljivoj i njenom tipu. Takođe se definišu metode za dodavanje oznake, odredišta, izvora, predaka i sledbenika instrukcijama, kao i metode za postavljanje upotrebe i definicije instrukcije.

Osim toga, u ovoj datoteci se nalaze i pomoćne funkcije kao što su findInstructionWithLabel i findInstructionAfterFunc, koje se koriste za pronalaženje odgovarajućih instrukcija na osnovu oznaka ili pozicije. Takođe, tu su i funkcije contains, koje proveravaju da li lista instrukcija ili promenljivih sadrži određenu instrukciju ili promenljivu.

Na kraju, u datoteci *IR.cpp* se nalazi i definicija funkcije operator<< koja omogućava ispisivanje informacija o promenljivoj ili instrukciji u standardni izlaz.

4.5 LexicalAnalysis.h

LexicalAnalysis.h je zaglavlje koje definiše klasu LexicalAnalysis koja se koristi za leksičku analizu programa napisanog u višem asamblerskom jeziku. Ova klasa koristi konačni automat definisan u klasi FiniteStateMachine za prepoznavanje i generisanje tokena iz programa.

Ova datoteka sadrži deklaracije sledećih funkcija članica klase LexicalAnalysis:

initialize:

Metoda za inicijalizaciju leksičke analize i konačnog automata.

Do:

Metoda koja vrši leksičku analizu.

readInputFile:

Metoda za čitanje ulaznog fajla koji sadrži programski tekst.

getNextTokenLex:

Metoda za dobijanje sledećeg leksičkog tokena iz izvornog koda programa.

getTokenList:

Metoda za dobijanje liste tokena pročitanih iz izvornog koda.

printTokens:

Metoda za ispisivanje liste tokena.

printLexError:

Metoda za ispisivanje greške u slučaju da postoji greška prilikom leksičke analize.

Osim toga, ovo zaglavlje sadrži i privatne članove klase kao što su:

inputFile:

Ulazni fajl koji sadrži programski tekst.

programBuffer:

Bafer koji sadrži sadržaj ulaznog fajla.

programBufferPosition:

Trenutna pozicija u baferu programa.

fsm:

Objekat konačnog automata.

tokenList:

Lista parsiranih tokena.

errorToken:

Atribut koji čuva informacije o grešnom tokenu, ukoliko dođe do greške prilikom parsiranja.

printMessageHeader:

Metoda koja se koristi za ispisivanje zaglavlja sa imenima kolona prilikom ispisivanja tokena.

4.5.1 LexicalAnalysis.cpp

U *LexicalAnalysis.cpp* datoteci, implementirane su funkcije članice klase LexicalAnalysis koje su deklarisane u zaglavlju *LexicalAnalysis.h.*

Evo kratak pregled funkcija koje su implementirane u *LexicalAnalysis.cpp*:

initialize:

Ova funkcija postavlja početno stanje za leksičku analizu, uključujući resetovanje pozicije u programskom baferu i inicijalizaciju konačnog automata.

Do:

Ova funkcija pokreće leksičku analizu. Iterira kroz ulazni programski kod, generiše tokene za svaki prepoznati token i dodaje ih u listu tokena. U slučaju greške, zaustavlja proces analize.

readInputFile:

Ova funkcija otvara i čita ulazni fajl koji sadrži programski kod. Učitani sadržaj se smešta u programski bafer.

getNextTokenLex:

Ova funkcija generiše sledeći leksički token iz programa. Koristi konačni automat za prepoznavanje tokena.

getTokenList:

Ova funkcija vraća listu tokena koja je generisana tokom leksičke analize.

printTokens:

Ova funkcija ispisuje listu tokena koja je generisana tokom leksičke analize. Koristi se za debagiranje ili prikazivanje rezultata analize.

printLexError:

Ova funkcija ispisuje grešku koja se desila tokom leksičke analize, ako postoji.

printMessageHeader:

Ova funkcija služi za ispisivanje zaglavlja pri prikazivanju listi tokena, čime se naglašava struktura prikaza.

4.6 Syntax Analysis.h

SyntaxAnalysis.h je zaglavlje koje definiše klasu Syntax Analysis i nekoliko pomoćnih funkcija za rad sa sintaksnom analizom.

Evo pregleda:

Syntax Analysis klasa:

Ova klasa se koristi za analizu tokena dobijenih iz leksičke analize. Ona ima metode za proveru grešaka, kreiranje promenljivih i instrukcija, kao i za izvršavanje same sintaksne analize.

Enumeracija SyntaxError:

Ovo je enumeracija koja definiše različite vrste sintaksnih grešaka koje mogu nastati tokom analize.

Pomoćne funkcije:

printError: Funkcija koja štampa tip sintaksne greške.

Članovi klase:

lex: Referenca na rezultate leksičke analize.

currentToken: Pokazivač na trenutni token koji se analizira.

instrs: Lista instrukcija.

reg_vars: Lista registarskih promenljivih.

mem_vars: Lista promenljivih za memorijske adrese.

label_vars: Lista labela.

const_vars: Lista konstantnih promenljivih.

err: Flag koji označava da li je došlo do greške tokom analize.

eof: Flag koji označava da li je dostignut kraj fajla.

next_instruction_has_label: Flag koji označava da li sledeća instrukcija treba da ima labelu.

Ova datoteka definiše osnovnu strukturu za sintaksnu analizu, uključujući metode za proveru grešaka, kreiranje promenljivih i instrukcija, kao i izvršavanje analize.

4.6.1 SyntaxAnalysis.cpp

SyntaxAnalysis.cpp implementira funkcionalnosti definisane u *SyntaxAnalysis.h*.

Evo pregleda ključnih delova koda:

Implementacija konstruktora i destruktora:

Konstruktor

inicijalizuje polja klase i postavlja pokazivač na trenutni token na početak liste tokena dobijenih iz leksičke analize.

Destruktor

briše dinamički alocirane objekte iz liste promenljivih, labela i instrukcija.

Metoda *Do*():

Ova metoda pokreće proces sintaksne analize. Prvo postavlja pokazivač na početak liste tokena, zatim poziva metodu *Q*() koja definiše početak pravila za analizu

Metode *printInstructions() i printVariables()*:

Ove metode štampaju instrukcije i promenljive dobijene tokom sintaksne analize.

Implementacije pomoćnih funkcija:

eat: Proverava i konzumira token odgovarajućeg tipa.

glance: Proverava tip trenutnog tokena.

regVariableExists, memVariableExists, labelExists: Proveravaju postojanje odgovarajućih promenljivih.

createVariable, findVariable, constVariable, findLabel: Kreiraju ili pronalaze promenljive ili labele.

Metode za proveru labela i funkcija:

checkLabels: Proverava da li su sve reference na labele definisane.

checkFunctions: Proverava postojanje tačno jedne početne funkcije.

Implementacija pravila za sintaksnu analizu:

Q, S, L, E: Ove metode implementiraju pravila za analizu jezika, koristeći rekurzivno spuštanje. U zavisnosti od tipa trenutnog tokena, odgovarajuća pravila se primenjuju.

Funkcija *printError*:

Ova funkcija se koristi za štampanje tipa sintaksne greške.

Ovaj fajl definiše ključnu logiku za sintaksnu analizu, uključujući provere grešaka, kreiranje instrukcija i promenljivih, kao i primenu pravila za analizu jezika.

4.7 LivenessAnalysis.h

Datoteka *LivenessAnalysis.h* definiše klasu LivenessAnalysis, koja se koristi za analizu života registarskih promenljivih i dodelu procesorskih registara.

Evo pregleda ključnih delova koda:

Konstruktor i metoda *Do()*:

Konstruktor prima referencu na objekat SyntaxAnalysis kako bi preuzeo instrukcije i promenljive iz sintaksne analize.

Metoda *Do()* pokreće sve analize života i dodelu resursa.

Metode *writeToFile(), printRegisters() i printGraph():*

writeToFile() kreira datoteku na datoj putanji i piše analizirani kod u nju ako je sve prošlo bez greške.

printRegisters() i printGraph() služe za štampanje registara i grafa interferencije na terminal

Pomoćne metode za analizu života:

liveness() pokreće analizu života.

setGraph() priprema graf interferencije.

resourceAllocation() vrši dodelu procesorskih registara.

Pomoćne metode za postavljanje prethodnika, sledbenika, korišćenih i definisanih promenljivih instrukcija:

setPredAndSucc() postavlja prethodnike i sledbenike instrukcija.

setUseAndDef() postavlja korišćene i definisane promenljive instrukcija.

Metode za rad sa grafom interferencije:

setInterference() postavlja vrednosti u grafu interferencije.

createSimplificationStack() kreira stek promenljivih prema rangiranju u grafu.

getColor() određuje koji registar treba dodeliti datoj promenljivoj na osnovu grafa interferencije i već dodeljenih registara.

Ova datoteka implementira ključne funkcionalnosti za analizu života registarskih promenljivih i dodelu procesorskih registara.

4.7.1 LivenessAnalysis.cpp

Datoteka *LivenessAnalysis.cpp* sadrži implementaciju metoda klase LivenessAnalysis.

Evo pregleda ključnih delova koda:

Konstruktor:

Konstruktor inicijalizuje promenljive i poziva metode *setPredAndSucc()* i *setUseAndDef()*.

Metoda *Do():*

Metoda *Do()* pokreće celokupnu analizu života, postavlja graf interferencije i vrši dodelu resursa.

Metoda *liveness():*

Ova metoda implementira algoritam analize života, iterira kroz instrukcije i ažurira ulazne i izlazne promenljive.

Metode za postavljanje prethodnika i sledbenika (*setPredAndSucc()*) i korišćenih i definisanih promenljivih (*setUseAndDef()*):

Ove metode postavljaju prethodnike, sledbenike, korišćene i definisane promenljive instrukcija.

Metode za kreiranje steka pojednostavljenja i određivanje boje (registra) promenljive:

createSimplificationStack() kreira stek promenljivih prema rangiranju u grafu.

getColor() određuje koji registar treba dodeliti datoj promenljivoj na osnovu grafa interferencije i već dodeljenih registara.

Metoda za postavljanje interferencije u grafu (*setInterference()*):

Ova metoda postavlja vrednosti u grafu interferencije.

Metode za štampanje registara i grafa interferencije (*printRegisters() i printGraph()*):

Ove metode služe za ispisivanje registara i grafa interferencije na terminal.

Metoda *writeToFile()*

Ova metoda kreira izlaznu datoteku i u nju upisuje analizirani kod.

Ova datoteka implementira celokupnu analizu života registarskih promenljivih i dodelu resursa na osnovu grafu interferencije.

4.8 FiniteStateMachine.h

Datoteka *FiniteStateMachine.h* definiše klasu FiniteStateMachine, koja predstavlja konačni automat za prepoznavanje tokena u niski karaktera.

Evo pregleda ključnih delova koda:

Definicija klase FiniteStateMachine:

Klasa FiniteStateMachine sadrži metod za pronalaženje sledećeg stanja na osnovu trenutnog stanja i karaktera tranzicije.

Metoda *getNextState()*:

Ovaj metod vraća broj sledećeg stanja na osnovu trenutnog stanja i karaktera tranzicije.

Metoda *initStateMachine()*

Inicijalizuje konačni automat, odnosno popunjava tabelu tranzicija sa odgovarajućim vrednostima.

Statički metod *getTokenType():*

Ovaj metod vraća tip tokena na osnovu broja stanja.

Članovi klase:

stateMachine: Mapa koja predstavlja tranzicionu matricu konačnog automata.

stateToTokenTable: Tabela koja mapira stanja na tipove tokena.

supportedCharacters: Niz podržanih karaktera.

stateMatrix: Matrica tranzicija između stanja.

Ova datoteka definiše osnovnu strukturu konačnog automata za analizu niski karaktera i njihovo prepoznavanje kao tokena.

4.8.1 FiniteStateMachine.cpp

FiniteStateMachine.cpp sadrži implementaciju metoda koji su deklarisani u *FiniteStateMachine.h.*

Evo ključnih delova:

Uključivanje Potrebnih Zaglavlja:

Datoteka uključuje "*FiniteStateMachine.h*" kako bi pristupila deklaracijama klase FiniteStateMachine i ostalim neophodnim elementima.

Upotreba Imenskog Prostora:

Koristi *using namespace std;* kako bi uključila ceo std imenski prostor u trenutni opseg. Ovo generalno nije preporučljivo u zaglavljima datoteka, ali je u redu u implementacionim datotekama.

Definicija Statičkih Članovskih Promenljivih:

Definiše se statičke članke promenljive *stateToTokenTable i supportedCharacters* koje su deklarisane u zaglavlju.

Definicija Statičke Funkcije Članice:

Statička funkcija članica *getTokenTyp*e je definisana ovde. Ona vraća tip tokena koji je povezan sa datim brojem stanja.

Definicija Funkcije Članice *initStateMachine:*

Ova funkcija inicijalizuje mapu stanja automata prolaskom kroz svako stanje i njegove tranzicije, i popunjava član *stateMachine.*

Definicija Funkcije Članice *getNextState:*

Ova funkcija uzima trenutno stanje i slovo tranzicije kao ulaz i vraća sledeće stanje na osnovu tranzicije. Proverava da li trenutno stanje postoji u mašini stanja i da li je slovo tranzicije validno.

U suštini, *FiniteStateMachine.cpp* pruža detalje implementacije funkcionalnosti konačnog automata definisane u zaglavlju datoteke.

4.9 main.cpp

main.cpp je glavni programski fajl koji predstavlja ulaznu tačku u aplikaciju. U ovom fajlu se obično nalazi funkcija *main()* koja pokreće izvršenje programa.

Evo kratkog opisa onoga što se dešava u *main.cpp*:

Uključivanje Potrebnih Zaglavlja:

Uključuju se standardna zaglavlja *<iostream> i <exception>,* kao i zaglavlje *LivenessAnalysis.h* koje sadrži deklaracije klase za analizu života.

Upotreba Imenskog Prostora:

Koristi se *using namespace std*; kako bi se uključio ceo std imenski prostor u trenutni opseg.

Funkcija *main():*

Ovde se definiše funkcija *main()*, koja je ulazna tačka programa. Programski tok počinje odavde.

Pokušaj-blok:

Koristi se try-catch mehanizam kako bi se uhvatile izuzetke koji se mogu pojaviti tokom izvršenja programa. Ako se bilo koji izuzetak desi tokom izvršenja, uhvaćen će biti u odgovarajućem catch bloku, gde se ispisuje odgovarajuća poruka o grešci.

Pokretanje Analiza:

Unutar glavne funkcije, prvo se vrši leksička analiza, zatim sintaksna analiza, i na kraju analiza života. Svaka od ovih analiza se izvršava pozivom metode *Do()* nad odgovarajućim objektima.

Ispis Rezultata:

Nakon izvršenja svake analize, program ispisuje odgovarajuće poruke o uspehu ili neuspehu, kao i rezultate analiza.

Uhvatanje Izuzetaka:

Ako se desi izuzetak tokom bilo koje analize, on se uhvata i ispisuje odgovarajuća poruka o grešci.

Na kraju, main() funkcija vraća 0 ako je program uspešno završen ili 1 ako se desila greška.

5. ZAKLJUČAK

5.1 Postignuti rezultati

Uspesno je izvršena leksička, sintaksna i analiza života.

Generisani su odgovarajući rezultati i ispisi tokom izvršenja analiza.

5.2 Moguća poboljšanja i budući rad

Optimizacija performansi algoritama analize, kako bi se smanjilo vreme izvršenja.

Proširenje funkcionalnosti programa dodavanjem podrške za dodatne operacije i konstrukcije jezika.

Poboljšanje korisničkog interfejsa i upravljanje greškama radi boljeg korisničkog iskustva.

Dodavanje automatskih testova radi provere ispravnosti implementacije.

Ovi koraci bi mogli da unaprede kvalitet i upotrebljivost programa u budućnosti.