Grafički prikaz matematičkih izraza

Anđelić, Ivana

Undergraduate thesis / Završni rad

2024

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Applied Sciences in Information Technology / Veleučilište suvremenih informacijskih tehnologija**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:289:355297

Rights / Prava: In copyright / Zaštićeno autorskim pravom.

Download date / Datum preuzimanja: **2025-02-22**

Repository / Repozitorij:

VSITE Repository - Repozitorij završnih i diplomskih radova VSITE-a

VELEUČILIŠTE SUVREMENIH INFORMACIJSKIH TEHNOLOGIJA

STRUČNI PRIJEDIPLOMSKI STUDIJ INFORMACIJSKIH TEHNOLOGIJA

Ivana Anđelić

ZAVRŠNI RAD

GRAFIČKI PRIKAZ MATEMATIČKIH IZRAZA

Zagreb, listopada 2024.

Veleučilište suvremenih informacijskih tehnologija
10000 Zagreb, Ulica Vjekoslava Klaića 7

Studij:	Stručni prijediplomski studij informacijskih tehnologija smjer baze podataka i web dizajn
Student:	**Ivana Anđelić**
Matični broj:	2013067

Zadatak završnog rada

Predmet:	Programiranje u C#
Naslov:	**Grafički prikaz matematičkih izraza**
Zadatak:	Napraviti aplikaciju koja će omogućiti interaktivni unos, izračunavanje i prikaz složenih matematičkih izraza koji u sebi mogu koristiti zagrade i osnovne matematičke funkcije.
Mentor:	mr. sc. Julijan Šribar, v. pred.
Zadatak uručen kandidatu:	2.10.2023.
Rok za predaju rada:	17.10.2024.
Rad predan:	_______________

Povjerenstvo:

Jurica Đurić, v. pred.	član predsjednik	______________________
mr. sc. Julijan Šribar, v. pred.	mentor	______________________
Mariza Maini, pred.	član	______________________

SADRŽAJ

1. UVOD.....7
2. Korištene tehnologije i alati.....9
   2.1. .NET okvir.....9
   2.2. Programski jezik C#.....9
      2.2.1. Metode.....11
      2.2.2. Svojstva.....11
      2.2.3. Pristupanje objektima.....11
      2.2.4. Sustav za upravljanje memorijom.....11
      2.2.5. Polimorfizam.....12
      2.2.6. Kategorije tipova podataka.....12
   2.3. Parsiranje.....13
   2.4. Oblikovni obrasci i obrazac interpretatora.....14
3. Implementacija Parsera i interpretatora.....15
   3.1. Parsiranje.....15
      3.1.1. Parsiranje od vrha prema dnu.....16
      3.1.2. Parsiranje od dna prema vrhu.....16
      3.1.3. Apstraktno sintaksno stablo.....16
      3.1.4. Shunting yard algoritam.....17
   3.2. Interpretator.....20
4. PRAKTIČNI RAD – PRIKAZ MATEMATIČKIH FUNKCIJA U PRAVOKUTNOM KOORDINATNOM SUSTAVU.....22
   4.1. Parsiranje algebarskih izraza.....22
   4.2. Implementacijski detalji metode Parse.....23
   4.3. GUI dio aplikacije.....29
5. ZAKLJUČAK.....31
SAŽETAK.....34
SUMMARY.....35

POPIS SLIKA

Slika 1. Proces parsiranja teksta (izvor: https://themightyprogrammer.dev/article/parsing)....12
Slika 2. Apstraktno sintaksno stablo za izraz `4 + 7 * 2`...16
Slika 3. GUI dio aplikacije...21
Slika 4. Prikaz pogreške u unesenom izrazu...29

POPIS KODOVA

Kôd 1. Kostur metode `Parse`....22

Kôd 2. Obrada lijeve i desne zagrade....23

Kôd 3. Procesuiranje predznaka ispred operanda....23

Kôd 4. Metoda `ProcessOperator`....24

Kôd 5. Stanje `ReadingVariable`....24

Kôd 6. Metoda `PushConstant`....25

Kôd 7. Metoda `PushFunction`....25

Kôd 8. Metoda `PushNamedConstant`....26

Kôd 9. Metoda `GetStateAfterOperator`....27

Kôd 10. Metoda `FinalExpression`....28

1. UVOD

Tema ovog završnog rada je parsiranje i izračunavanje matematičkih izraza, a u okviru rada je napravljena aplikacija za grafički prikaz matematičkih izraza u pravokutnom koordinatnom sustavu. Sama problematika se uglavnom sastojala u implementaciji parsera, dijela koji je odgovoran za interpretaciju korisnikovog unosa. Ne radi se samo o jednostavnim funkcijama, već o kompleksnim izrazima koji mogu sadržavati više matematičkih funkcija, konstanti i varijabli. Bilo je potrebno analizirati i raščlaniti cijeli izraz koji korisnik unese, uključujući konstante, operatore množenja, dijeljenja, zbrajanja, oduzimanja i potencije, funkcije, ostale alfanumeričke znakove koji mogu predstavljati varijable, lijevu i desnu zagradu. Također je bilo potrebno implementirati rukovanje neispravnim korisnikovim unosom.

Jedan od glavnih problema je bilo rješavanje hijerarhije operacija. Za to je iskorišten algoritam ranžirnog kolodvora (engl. *shunting yard algorithm*), postupak za parsiranje aritmetičkih izraza, logičkih izraza ili njihove kombinacije, zadanih infiksnom notacijom. Algoritam ranžirnog kolodvora se zasniva na rukovanju stogom (engl. *stack*). Bilo je potrebno voditi računa o stanju u kojem se nalazi parser, što se postiglo implementacijom automata s konačnim brojem stanja (engl. *state machine*). Primjerice, stanje može biti preskakanje znakova praznine prije operatora, čitanje imena varijable, čitanje imena funkcije ili čitanje vrijednosti konstante. Prema tome je bilo potrebno predvidjeti i buduće moguće stanje te na stog spremiti odgovarajući objekt. Da bi se izračunao rezultat parsiranja, implementiran je odgovarajući interpretator.

Od matematičkih funkcija podržane su sve trigonometrijske funkcije, eksponencijalna, logaritamska te funkcija apsolutne vrijednosti. Podržana je i kompozicija funkcija, tj. definiranje argumenta funkcije drugom funkcijom. Sam projekt praktičnog rada sadrži četiri podprojekta: biblioteke koja implementira parser i interpretator, biblioteke s kontrolom za prikaz pravokutnog koordinatnog sustava, same GUI aplikacije te projekta s jediničnim testovima. GUI je realiziran korištenjem biblioteke Windows Forms. U glavnu formu se upisuje matematički izraz koji se želi prikazati, parametri za kontrolu prikaza te se prikazuje koordinatni sustav s grafom izraza. Jediničnim testovima su se testirale funkcije i klase korištene u implementaciji parsera i interpretatora.

Motivacija za izbor ove teme bila je želja da se nauči kako implementirati takav, naizgled jednostavan zadatak. Inspiracija za to je pronađena u laboratorijskim vježbama iz matematičke analize gdje se, unoseći neku funkciju u programu Sage, dobio njen grafički prikaz. Otada je postojalo zanimanje kako se ta funkcionalnost postiže, te je ovim radom,

koristeći programski jezik C#, upravo takva funkcionalnost realizirana. Manjim dijelom korištena je već gotova funkcionalnost koju ovaj jezik nudi, a većim dijelom stvari su pisane ispočetka. Tako su implementirane posebne klase, a crtanje grafa u GUI dijelu u potpunosti je prilagođeno potrebama ovog projekta.

U drugom poglavlju objašnjene su tehnologije i alati korišteni za izradu aplikacije za crtanje funkcije u koordinatnom sustavu te osnovni pojmovi koji se odnose na parsiranje i interpretator. U trećem poglavlju opisani su implementacijski detalji parsera i interpretatora te algoritam ranžirnog kolodvora koji je korišten u izradi ovog projekta. U četvrtom poglavlju obrađena je konkretna primjena parsera i interpretatora u aplikaciji te su opisani važni dijelovi koda i GUI dio aplikacije.

2. KORIŠTENE TEHNOLOGIJE I ALATI

U ovom poglavlju bit će objašnjene tehnologije u kojima je ovaj projekt realiziran.

2.1. .NET okvir

.NET je okvir (engl. *framework*) za razvoj svih vrsta aplikacija. Trenutno je .NET platforma otvorenog koda (engl. *open source*) koju održava Microsoft i .NET zajednica.

Programi napisani u izvornom kodu koji se izvode na .NET-u se prevode u međujezik (IL – engl. *Intermediate Language*) te se u tom obliku distribuiraju korisnicima. CLR (engl. *Common Language Runtime*) je izvedbena okolina .NET okvira zadužena za izvođenje programa prevedenog u međujezik. IL kôd je potpuno neovisan o programskom jeziku te se iz kôdova pisanih u različitim jezicima dostupnima na .NET platformi, kao što su C#, Visual Basic i F#, može generirati potpuno identičan IL kôd. Iz toga proizlazi i velika prednost .NET platforme: mogućnost višejezičnog pisanja aplikacija, naravno uz uvjet da postoji odgovarajući Intermediate Language kompajler.

U samim počecima .NET je primarno razvijen kao platforma za Windows operacijski sustav pod nazivom .NET Framework. Alternativna platforma, .NET Core, pojavila se 2016. godine i od tada je .NET multiplatformski sustav, što znači da je podržan na Windows, Linux i macOS operacijskim sustavima te je uz stare značajke dodana i nova funkcionalnost. Trenutna verzija, .NET 8 (studeni 2023.) ujedinjuje svu funkcionalnost i omogućava razvoj web, mobilnih, cloud i IoT aplikacija na raznim platformama. Desktop aplikacije je i dalje moguće razvijati samo na Windows operacijskim sustavima.

Proces razvoja je također olakšan time što uz .NET dolazi i .NET standardna biblioteka koja sadrži već gotove komponente koje se mogu iskoristiti u velikom broju aplikacija. Osim multiplatformske podrške i izbora programskih jezika, prednosti .NET-a su automatizacija raznih procesa, kao što je automatsko oslobađanje memorije (engl. *garbage collection*), provjera kôda i testiranje, fleksibilnost. Zbog toga su aplikacije u .NET-u jednostavne za održavanje.

2.2. Programski jezik C#

Programski jezik C# je objektno orijentirani programski jezik opće namjene koji obuhvaća generičke, deklarativne, funkcionalne, komponentno orijentirane i objektno orijentirane programske discipline. Razvila ga je tvrtka Microsoft unutar .NET sustava te ga je odobrio i standardizirao ISO, međunarodna organizacija za standardizaciju (engl. *International Organization for Standardization*). C# je također jedan od programskih jezika dizajniran za

CLI (engl. *Common Language Infrastructure*), infrastrukturu koja uključuje CLR i biblioteku klasa neophodnu za pisanje i izvođenje programa pisanih u jeziku C#.

Osnovna svojstva programskog jezika C# su:

- On je jednostavan, moderan, objektno orijentirani jezik opće namjene.
- Podržava programsko inženjerske principe, kao što su stroga tipizacija i automatsko upravljanje memorijom.
- To je portabilan jezik, što znači da se program napisan u C# može pokrenuti na različitim platformama.
- Podržava internacionalizaciju i lokalizaciju (engl. *internationalization and localization*), što omogućava jednostavnu lokalizaciju resursa (npr. tekstova poruka) za bilo koji govorni jezik.
- C# je predviđen za pisanje aplikacija u poslužiteljskim ili klijentskim sustavima, kao i za ugradbene sustave (engl. *embedded systems*).
- C# je ekonomičan jezik s obzirom na memorijske i procesorske resurse, iako se ne može usporediti s jezicima kao što su C, C++ ili asembler.

Jezik C# podržava strogu tipizaciju, implicitno tipizirane deklaracije varijabli ključnom riječi `var` te tipizirane nizove s ključnom riječi `new[]` koju slijedi inicijalizator kolekcije. C# također podržava logički podatkovni tip `bool`. Petlja `while` i naredba grananja `if` zahtijevaju izraz čiji rezultat mora biti tipa `bool` (istina ili laž). Jezik C++ također ima tip `bool`, no zbog implicitnih pretvorbi u taj tip moguće je, na primjer, uvjetima naredbi `if` navesti izraz koji kao rezultat daje cijeli broj ili pokazivač (engl. *pointer*) a da prevoditelj ne prijavi pogrešku. C# ne dopušta ovakve konverzije te prisiljava programera da koristi samo izraze koji kao rezultat vraćaju `bool`, čime se smanjuje mogućnost programskih pogrešaka.

Što se tiče tipske sigurnosti, jedine implicitne konverzije koje su dopuštene su iz izvedenog tipa u bazni tip. Operatorima implicitne pretvorbe može se skup implicitnih pretvorbi proširiti. Na taj su način omogućene pretvorbe između ugrađenih brojčanih tipova, na primjer iz tipa `int` u tip `double`. Implicitne konverzije između tipa `bool` i tipa `int` nisu dozvoljene, kao ni između članova enumeracija i tipa `int`, osim nule koja može biti implicitno konvertirana u bilo koji enumerabilni tip.

Jezik C# također ne dopušta globalne varijable ili funkcije. Sve metode i varijable moraju biti definirane unutar klase. Statički članovi javnih klasa mogu nadomjestiti globalne varijable ili funkcije. Lokalne varijable ne mogu skrivati istoimene varijable unutar bloka, za razliku od programskih jezika C ili C++ koji to omogućavaju.

2.2.1. Metode

Kao i u drugim sintaktički sličnim jezicima, poput C++ ili C, potpis metode je deklaracija koja se sastoji redom od ključne riječi koja označava vidljivost (npr. `private`), povratnog tipa (npr. `int` ili `void`), imena metode te parametara unutar zagrada koji su razdvojeni zarezom, a svaki se sastoji od tipa, formalnog imena i opcionalnih podrazumijevanih vrijednosti koje se koriste kada pripadajući argumenti nisu eksplicitno zadani pri pozivu metode.

Slično kao u C++-u, a za razliku od Jave, programer mora koristiti ključnu riječ `virtual` ako želi da njegova metoda može biti prepisana u podklasi.

Metode proširenja (engl. *Extension methods*) u C#-u su statičke metode s posebnim potpisom koje programeri mogu dodjeljivati tipovima čak i kada nemaju dostupan izvorni kôd tipa. Te metode se onda mogu pozivati nad instancama objekta, kao da su definirane u originalnoj tablici klasnih metoda.

2.2.2. Svojstva

C# podržava klase sa svojstvima (engl. *property*). Svojstva su sintaktički šećer za opći obrazac u kojem metode pristupnik (engl. *getter*) i mutator (engl. *setter*) enkapsuliraju operacije nad poljem unutar klase. Prilikom poziva, svojstvu se može pristupiti pomoću pristupnika umjesto opširnijeg poziva metoda.

2.2.3. Pristupanje objektima

U jeziku C# objektima se pristupa preko referenci koje uvijek pokazuju na validni objekt ili imaju vrijednost `null` reference te je nemoguće koristiti referencu na objekt koji je obrisan ili u sebi sadrži memorijsku adresu kojoj se ili ne smije pristupiti ili pokazuje na nasumični memorijski blok. Pokazivači (engl. *pointer*) na proizvoljne memorijske adrese mogu biti korišteni samo unutar blokova koji su specifično označeni kao *nesigurni* (engl. *unsafe*) i programi s takvim kodom zahtijevaju da se u postavkama projekta uključi opcija da su nesigurni blokovi dozvoljeni. Kôd koji nije označen kao nesiguran i dalje može sadržavati i manipulirati pokazivačima kroz `System.IntPtr` tip, ali ih ne može dereferencirati.

2.2.4. Sustav za upravljanje memorijom

Memoriju koju zauzimaju objekti na hrpi (engl. *heap*) nadzire sustav za upravljanje smećem (engl. *garbage collector*) pa se ta memorija naziva upravljanom memorijom (engl. *managed memory*) i ona ne može biti eksplicitno oslobođena iz programskog kôda. Sustav za upravljanje smećem prati reference na objekte i kada ne postoji više niti jedna izravna ili

posredna referenca na objekt, proglašava tu memoriju slobodnom za nove objekte. Na taj način oslobađa programera od odgovornosti za oslobađanjem alocirane memorije koja mu više nije potrebna i sprečava problem curenja memorije (engl. *memory leak*).

2.2.5. Polimorfizam

Polimorfizam dolazi od grčkih riječi *poly* koja znači mnogo i *morph* koja znači forma. Polimorfizam omogućava da se istoimena operacija izvodi na različite načine.
Postoje dva tipa polimorfizma u jeziku C#, a to su statički i dinamički polimorfizam. Primjer statičkog polimorfizma je preopterećenje metoda (engl. *method overloading*), kada metode imaju isto ime, ali s različitim parametrima, a odluka o tome koja će se metoda pozvati događa se za vrijeme kompajliranja (engl. *compile time*). Kod dinamičnog polimorfizma bazna klasa sadrži potpis metode označenom ključnom riječi `virtual` koja označava da sve klase koje se izvedu iz te bazne klase moraju implementirati tu metodu. Implementacija se može razlikovati od klase do klase, a odluka o tome koja će se metoda pozvati donosi se prilikom izvođenja (engl. *runtime*) te ovisi konkretnom objektu klase nad kojim je metoda pozvana.

2.2.6. Kategorije tipova podataka

Jezik C# podržava dvije kategorije tipova podataka:

- referentne tipove i
- vrijednosne tipove.

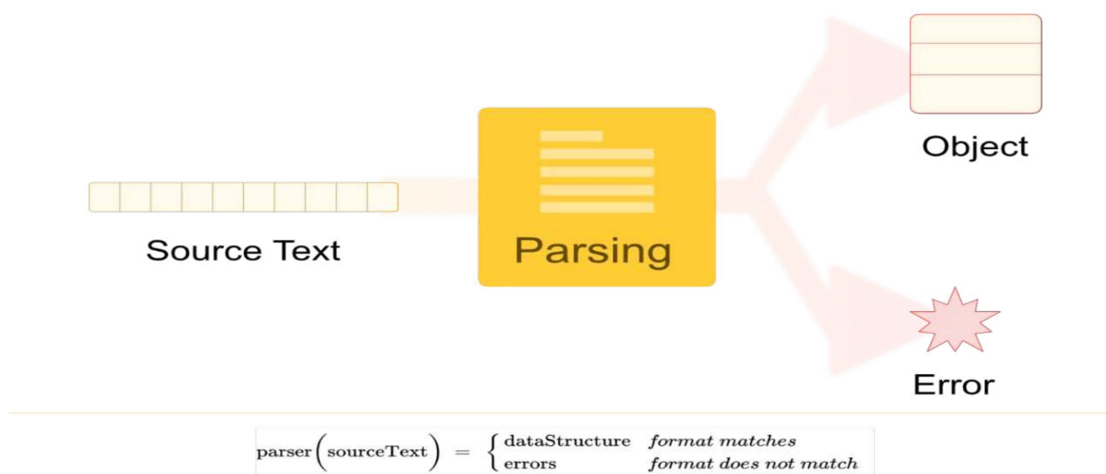
Instance vrijednosnih tipova nemaju referentni identitet ni referentnu usporednu semantiku. Usporedbe jednakosti i nejednakosti vrijednosnih tipova uspoređuju stvarne vrijednosti podataka instanci, osim ako su pridruženi operatori usporedbe preopterećeni. Vrijednosni tipovi nasljeđuju *System.ValueType* i ne mogu se nasljeđivati, ali mogu implementirati sučelja. Podrazumijevani konstruktor bez parametara će inicijalizirati članove na unaprijed određenu vrijednost: na nulu za vrijednosne tipove, odnosno `null` referencu za referentne tipove. Primjeri vrijednosnih tipova su svi primitivni tipovi, kao `int`, `float`, `char` te `enum` (enumeracije) i korisnički definirane strukture.
Za razliku od vrijednosnih tipova, referentni tipovi imaju referencijalni identitet, što znači da je vrijednost referentnog tipa različita od drugih instanci, čak i ako objekti na koji reference pokazuju imaju identičan sadržaj. To se odražava u usporedbi jednakosti i nejednakosti za referentne tipove, koja testira po referentnoj, a ne strukturalnoj jednakosti, osim ako su

odgovarajući operatori preopterećeni. Primjeri referentnog tipa su klasa `Object`, iz koje su izvedene sve druge klase, klasa `String` te `Array`, koji je bazna klasa za sve C# nizove.

2.3. Parsiranje

Algoritmi za parsiranje se uobičajeno koriste kako bi se parsirao neki tekstualni izraz. Zbog toga se kaže kako su algoritmi za parsiranje napravljeni za jezičnu gramatiku (engl. *grammar*).

Parser koristi gramatiku kako bi analizirao niz znakova i simbola te ih pretvorio u strukturalnu reprezentaciju koja se može dalje procesuirati [3]. Gramatika je skup pravila koja određuju kako će se slagati jezični znakovi da bi formirali valjanu rečenicu. Na primjer, hrvatska gramatika omogućava da slova 't', 'i' jedno za drugim formiraju valjanu riječ 'ti'. Ako se na nju dodaju znakovi ,' ','s', 'i', ' ', 's','u','p','e','r', formira se validna rečenica 'ti si super'. Parser je algoritam koji analizira niz validnih riječi (engl. *token*) i odlučuje je li taj niz ispravan za zadanu gramatiku (Error: Reference source not found). Među ostalim, parser se koristi tijekom prevođenja izvornog kôda. Kompajler uzima niz znakova na ulazu, na primjer datoteku s programskim kodom koji se mora kompajlirati. Prvi korak u procesu kompajliranja je leksička analiza koja na osnovu cijelog teksta generira niz tokena. Taj niz se predaje parseru, koji gradi hijerarhijsku strukturu podataka, najčešće kao stablo. Pri tome, parser može donijeti odluku o odbacivanju takvog ulaza ako on nije usklađen sa zadanom gramatikom. Ako su tokeni postavljeni jedan iza drugoga u validnom redoslijedu, tada će niz biti prihvaćen.



Slika 1. Proces parsiranja teksta (izvor: https://themightyprogrammer.dev/article/parsing)

Budući da gramatika može biti komplicirana [2], algoritam koji odlučuje o validnosti niza znakova ili tokena ne mora biti trivijalan. U ovom projektu, gramatika je definirana preko matematičkih operatora, varijabli, konstanti i funkcija. Primjer matematičke gramatike je da operator zbrajanja mora i s lijeve i s desne strane imati konstantu, varijablu, funkciju ili neki drugi izraz što bi predstavljalo jednostavni izraz. Složeni izraz se može sastojati od operatora i jednostavnih izraza ili drugih složenih izraza s lijeve i desne strane. Dakle, izraz može biti samo varijabla, izraz oblika `a + b` ili se, na primjer s desne strane operatora može nalaziti funkcija, a s lijeve varijabla.

2.4. Oblikovni obrasci i obrazac interpretatora

U softverskom inženjerstvu, oblikovni obrazac je u općenitom smislu rješenje za neki problem koji se često pojavljuje u određenom kontekstu [1]. Oblikovni obrasci mogu ubrzati razvojni proces jer pružaju testirane i dokazane razvojne paradigme [5]. Također, omogućuju fleksibilniji dizajn i promjenu ponašanja tijekom izvođenja. Ponašanje kôda se mijenja ovisno o ulaznim podacima pa je za isti kôd moguće ostvariti različite rezultate. Fleksibilan softverski dizajn zahtijeva da se u obzir uzmu problemi koji mogu postati vidljivi tek kasnije u implementaciji. Novonapisani kôd može imati skrivene probleme koji se mogu otkriti tek kasnije, a neki od njih nisu zanemarivi. Korištenje oblikovnih obrazaca takve probleme svodi na minimum te omogućuje bolje razumijevanje kôda za programere i arhitekte koji su upoznati s konkretnim obrascima.

Po definiciji, obrazac se mora uvijek iznova implementirati u svakoj aplikaciji koja ga koristi. S obzirom da neki autori smatraju to korakom unazad u ponovnoj iskoristivosti kôda, arhitekti su odlučili implementirati pojedine obrasce u komponente. Ponekad je oblikovne obrasce ipak teško primijeniti u širokom spektru problema, iako donose neka generalna rješenja.

U ovom radu korišten je oblikovni obrazac interpretatora (engl. *interpreter*), koji definira način kako interpretirati neku jezičnu gramatiku ili izraz [4]. Da bi to bilo moguće, oblikovni obrazac podrazumijeva implementaciju određenih klasa koje predstavljaju jezičnu gramatiku ili, u slučaju ovog projekta, matematičke izraze. Oblikovni obrazac omogućava hijerarhijski poredak tih klasa kako bi se mogli interpretirati kompleksniji izrazi.

3. IMPLEMENTACIJA PARSERA I INTERPRETATORA

U ovom poglavlju će biti objašnjene glavne teoretske značajke parsera i interpretatora.

3.1. Parsiranje

Parsiranje matematičkih izraza je proces analize i interpretacije nizova simbola, to jest matematičkih izraza, kao što su funkcije, konstante te operatori zbrajanja, oduzimanja, množenja i dijeljenja. Cilj parsiranja je pretvoriti ove simbole u strukturu koja omogućava njihovu interpretaciju.

Koraci u parsiranju matematičkih izraza su:

1. Tokenizacija (Leksička analiza), proces rastavljanja unesenog izraza na manje dijelove zvane tokeni. Na primjer, izraz `4 + 7 * 2` se rastavlja na tokene `4`, `+`, `7`, `*`, `2`. Tokeni obično predstavljaju brojeve, operatore (`+`, `-`, `*`, `/`), zagrade i druge simbole.
2. Parsiranje, koje uključuje analizu niza tokena koja se zasniva na gramatici jezika, odnosno pravilima što su u slučaju ovog projekta klase koje predstavljaju matematičke izraze, te njihovo organiziranje u apstraktno sintaksno stablo (engl. *Abstract Syntax Tree* – AST).
3. Izgradnja apstraktnog sintaksnog stabla. Parseri grade takozvano parser stablo, podatkovnu strukturu koja se grana nalik na stablo i čiji su listovi završni tokeni gramatike (engl. *terminal*), a posredni čvorovi u stablu su nezavršeni tokeni (engl. *non-terminal*). Apstraktno sintaksno stablo je strukturalna reprezentacija matematičkog izraza. Svaki čvor stabla predstavlja operaciju ili operator, a strukturu stabla čini hijerarhija matematičkih operacija.
4. Evaluacija apstraktnog sintaksnog stabla. Nakon što je apstraktno sintaksno stablo izgrađeno, može se evaluirati kako bi se dobio rezultat matematičkog izraza. Evaluacija se obično radi rekurzivno, tako da se evaluira svako podstablo i na kraju izračuna vrijednost izraza. Poznata je činjenica da se algoritam rekurzije može implementirati u iterativnoj proceduri, a tipično se za to koristi struktura stoga (engl. *LIFO – Last In First Out*).

Postoje dva glavna pristupa parsiranju:

1. Od vrha prema dnu (engl. *Top-down*) gdje parsiranje počinje od najvišeg nivoa gramatike odnosno pravila i postupno se spušta ka tokenima.
2. Od dna prema vrhu (engl. *Bottom-up*) gdje parsiranje počinje od tokena i postupno gradi apstraktno sintaksno stablo.

3.1.1. Parsiranje od vrha prema dnu

Parsiranje od vrha prema dnu je metoda parsiranja koja počinje od najviše razine gramatike (korijena) i postupno se spušta prema terminalima (tokenima) korisnikova unosa te pritom pronalazi odgovarajuću strukturu sintaksnog stabla prema pravilima gramatike. Parseri od vrha prema dnu polaze od toga da je prvi vidljivi simbol u ulaznom nizu list stabla te grade stablo donoseći odluke o tome koji je roditeljski čvor tog lista (engl. *parent*), sve dok ne stignu do korijena stabla. Ovaj pristup se često koristi za parsiranje matematičkih izraza zbog njegove jednostavnosti i prirodne rekurzivne strukture matematičkih izraza. Parsiranje od vrha prema dnu koristi rekurzivno spuštanje, gdje se za svaki čvor u gramatici koristi odgovarajuća rekurzivna funkcija. Rekurzivna funkcija troši dijelove ulazne sekvence kako bi stvorila odgovarajuće sintaksno stablo prema pravilima gramatike.

Prednost parsiranja od vrha prema dnu je jednostavnost implementacije. Nedostaci su moguća neefikasnost za složenu gramatiku, posebno ako pravila gramatike nisu prilagođena za rekurzivno spuštanje te beskonačna rekurzija.

3.1.2. Parsiranje od dna prema vrhu

Parsiranje od dna prema vrhu je metoda parsiranja u kojoj proces sintaksne analize počinje od najmanjih jedinica to jest terminala (tokena) i postupno se diže prema čvorovima, sve dok se ne dobije početni čvor gramatike koji jest cijeli ulazni izraz. Parsiranje od dna prema vrhu počinje analizom tokena i pokušava prepoznati manje strukture koje odgovaraju pravilima gramatike, a zatim ih kombinira u veće strukture. Ovaj proces se nastavlja sve dok nije formiran korijen sintaksnog stabla, to jest cijeli ulazni izraz. Za ovakav način parsiranja koristi se stog za praćenje trenutnog stanja parsiranja. Simboli koje je parser prepoznao kao dio izraza nalaze se na stogu, a novi simboli (tokeni) se dodaju na stog kako parser napreduje kroz ulaznu sekvencu. U kontekstu matematičkih izraza, parsiranje od dna prema vrhu je popularno zbog svoje efikasnosti i sposobnosti da se nosi sa složenijim gramatikama.

3.1.3. Apstraktno sintaksno stablo

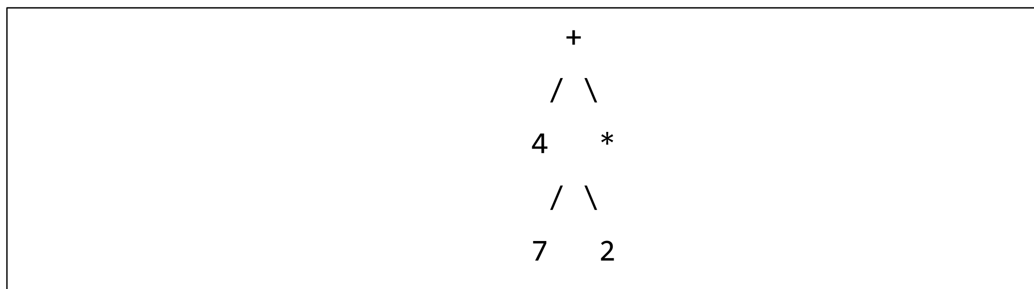
Apstraktno sintaksno stablo (engl. *Abstract Syntax Tree*) predstavlja strukturu ulaznog niza , tj. unesenog izraza. Prilikom parsiranja matematičkih izraza, AST se formira prema hijerarhiji operacija i njihovih operanada te čini logičku strukturu izraza.

Ključne karakteristike AST-a su:

1. Čvorovi (engl. *nodes*), koji mogu biti:

- Unutrašnji čvorovi (engl. *internal nodes*): predstavljaju operacije (npr. aritmetičke operatore kao što su `+`, `-`, `*`. `/`). Svaki unutrašnji čvor ima jednu ili više grana (engl. *child*) koje vode do njihovih operanada.
- Listovi (engl. *leaves*): predstavljaju operande (brojevi, varijable). Listovi su čvorovi bez grana.

2. Struktura i hijerarhija:
   - AST se gradi ovisno o prioritetu (engl. *precedence*) i asocijativnosti. Na primjer, u izrazu `4 + 7 * 2` množenje (`*`) ima viši prioritet od zbrajanja (`+`) pa će `*` biti u podstablu koje je grana to jest dijete čvora `+`.
   - AST jasno prikazuje redoslijed izvođenja operacija, što znači da se operacije na višim razinama izvode nakon operacija na nižim razinama (Slika 2).



Slika 2. Apstraktno sintaksno stablo za izraz `4 + 7 * 2`

3. Apstrakcija: AST je apstrakcija konkretne sintakse. Na primjer, zagrade su potrebne da bi se ispravno interpretirao izraz `(4 + 7) * 2`, ali u AST-u one nisu izravno prikazane. Umjesto toga, hijerarhija stabla se brine o prioritetu operacija.

AST je ključni koncept u parsiranju matematičkih izraza, kao i u širem kontekstu parsiranja programskog koda. AST predstavlja logičku strukturu izraza, što čini osnovu za daljnju evaluaciju, optimizaciju ili prevođenje unesenog niza.

3.1.4. Shunting yard algoritam

Algoritam ranžirnog kolodvora (engl. *Shunting Yard algorithm*) je metoda parsiranja aritmetičkih ili logičkih izraza ili njihove kombinacije, zadanih infiksnom notacijom [6]. Infiksna notacija se obično koristi u aritmetičkim i logičkim formulama i izrazima kada se operator nalazi između operanada, kao što je znak plusa u izrazu `2 + 2`. Algoritam ranžirnog kolodvora može generirati postfiksni niz, poznat kao *Reverse Polish Notation* (RPN), ili apstraktno sintaksno stablo. RPN je matematička notacija u kojoj operatori slijede svoje

operande, npr. `3 4 +`. Sufiksna notacija je pogodnija za evaluaciju izraza jer ne zahtijeva eksplicitne zagrade da bi se odredio redoslijed operacija.

Algoritam ranžirnog kolodvora je izumio Edsger Dijkstra i dobio je ime po grananju željezničkih pruga. Kao i evaluacija RPN-a, ovaj algoritam bazira se na stogu. Algoritam se sastoji od dvije glavne sastavnice:

1. Stog (engl. *stack*): koristi se za čuvanje operatora i zagrada dok se ne nađe njihov odgovarajući operand.
2. Izlazna lista: sadrži krajnju notaciju izraza u sufiksnom obliku.

Algoritam prolazi kroz ulazni niz simbola (tokena) jedan po jedan i na osnovu vrste svakog simbola, bilo da je operator, zagrada ili broj, odlučuje o daljnjim koracima – da li će ga staviti na stog, dodati u izlaznu listu ili primijeniti operaciju nad operandima iz stoga.

Algoritam ranžirnog kolodvora sastoji se od sljedećih koraka:

1. Čitanje tokena:
    - Broj ili operand se odmah dodaju u izlaznu listu.
    - Operator: ako je na vrhu operator koji ima veći ili jednak prioritet od trenutnog operatora, operator na vrhu se uklanja sa stoga i dodaje u izlaznu listu te se nakon toga trenutni operator stavlja na stog.
    - Lijeva zagrada se stavlja na stog.
    - Desna zagrada znači da se svi operatori uklanjaju sa stoga i dodaju u izlaznu listu sve do lijeve zagrade. Lijeva zagrada se uklanja sa stoga, ali se ne dodaje u izlaznu listu.
    - Kraj izraza: svi tokeni su obrađeni, a preostali operatori se uklanjaju sa stoga i dodaju u izlaznu listu.
2. Prijelaz na sljedeći token: algoritam nastavlja sa sljedećim tokenom dok se ne potroši cijeli ulazni niz.

Algoritam ranžirnog kolodvora prikazan na konkretnom primjeru izraza `2 + 4 * (7 - 3)`:

1. Početak:
    - Izlazna lista je prazna: [ ],
    - Stog je prazan: [ ].
2. Token `2`:
    - Broj se dodaje u izlaznu listu.
    - Izlazna lista: [`2`]

- Stog: [ ]

3. Token +:
   - Operator zbrajanja se stavlja na stog.
   - Izlazna lista: [2]
   - Stog [+]
4. Token 4:
   - Broj se dodaje u izlaznu listu.
   - Izlazna lista: [2, 4]
   - Stog: [+]
5. Token *:
   - Operator množenja se stavlja na stog jer ima viši prioritet od operatora zbrajanja koji se već nalazi na stogu.
   - Izlazna lista: [2, 4]
   - Stog: [+, *]
6. Token (:
   - Lijeva zagrada se stavlja na stog jer ima viši prioritet od operatora množenja koji se nalazi na vrhu stoga.
   - Izlazna lista: [2, 4]
   - Stog: [+, *, (]
7. Token 7:
   - Broj se dodaje u izlaznu listu.
   - Izlazna lista: [2, 4, 7]
   - Stog: [+, *, (]
8. Token -:
   - Operator oduzimanja se stavlja na stog.
   - Izlazna lista: [2, 4, 7]
   - Stog: [+, *, (, -]
9. Token 3:
   - Broj se dodaje u izlaznu listu.
   - Izlazna lista: [2, 4, 7, 3]
   - Stog: [+, *, (, -]

10. Token):

- Desna zagrada znači da se svi operatori sa stoga do lijeve zagrade trebaju ukloniti.
- Izlazna lista: [2, 4, 7, 3, -]
- Stog: [+, *]

11. Kraj izraza:

- U izlaznu listu se prebacuju preostali operatori sa stoga.
- Izlazna lista: [2, 4, 7, 3, -, *, +]
- Stog: []

Krajnja izlazna lista [2, 4, 7, 3, -, *, +] predstavlja izraz u sufiksnoj notaciji, a izračunava se na principu stoga:

1. `2 4 7 3 -`: `7 - 3` daje 4 pa izlazna lista postaje [2, 4, 4, *, +],
2. `4 * 4` daje 16 pa izlazna lista postaje [2, 16, +],
3. `2 + 16` daje 18 i to je konačan rezultat.

3.2. Interpretator

Ključni koncepti oblikovnog obrasca interpretatora su gramatika, parser, metoda `interpret`, kontekst i konkretne klase. Gramatika je sastavni dio oblikovnog obrasca i ona definira pravila jezika što uključuje terminalne i neterminalne simbole. Parser je aparat za analizu tj. dio interpretatora koji koristi gramatiku kako bi analizirao i interpretirao ulazni niz znakova ili simbola te ih pretvara u strukturnu reprezentaciju (AST) ili neku drugu internu strukturu. Metoda `interpret` je definirana unutar bazne apstraktne klase, a svaka klasa izvedena iz apstraktne klase implementira ovu metodu na poseban način. Kontekst označava lokalne ili globalne varijable u koje se sprema neko stanje. Konkretne klase predstavljaju određeno pravilo gramatike, a svaka od ovih klasa implementira metodu `interpret` s obzirom na pravilo gramatike koje predstavlja.

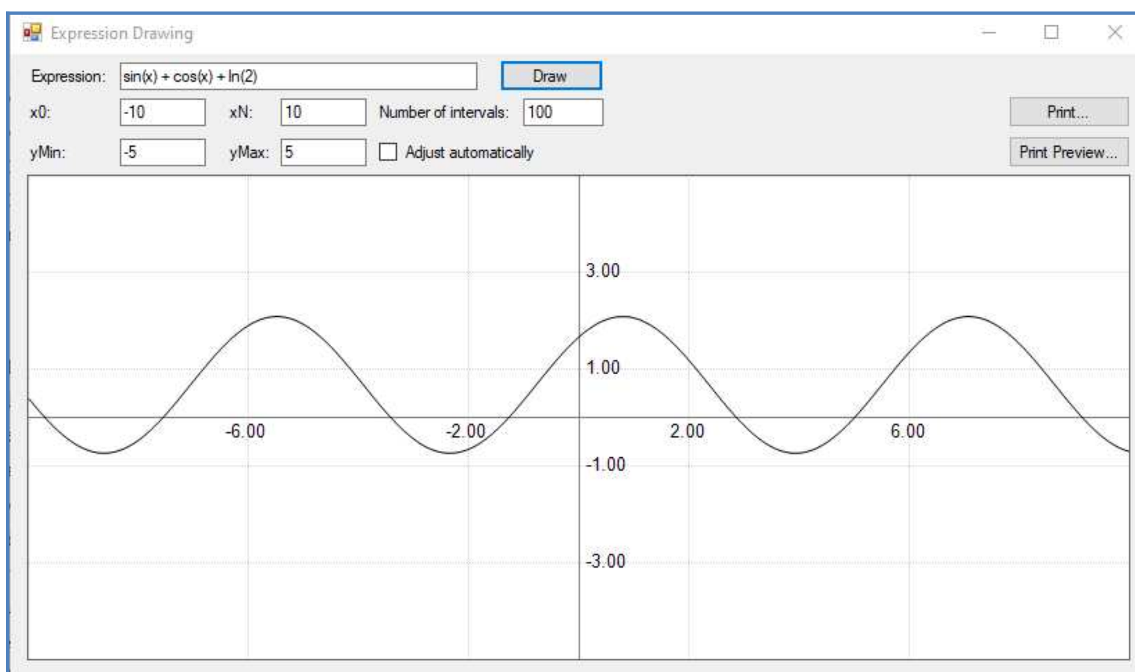
U okviru ovog projekta implementiran je interpretator za matematičke izraze. Interpretator za matematičke izraze prima kao ulaz matematičke izraze i izvršava operacije koje ti izrazi obuhvaćaju te obrađuje vraćeni rezultat. Za razliku od kompajlera koji prevodi kod u međujezik ili izvršni kod, interpretator analizira izraz u realnom vremenu. To omogućava da se isti programski kôd može koristiti za različite ulazne izraze.

Uobičajeni koraci u radu interpretatora za matematičke izraze su:

1. Tokenizacija: matematički izraz se dijeli na osnovne komponente tj. tokene. Tokeni mogu biti operatori (`+`, `-`, `*`, `/`), brojevi, funkcije (`sin`, `cos` …) i zagrade.
2. Parsiranje: u fazi parsiranja interpretator koristi parser koji analizira i obrađuje tokene te na temelju toga gradi apstraktno sintaksno stablo ili neku drugu internu strukturu. Parser pomoću AST-a također analizira prioritet operacija, npr. množenje ima veći prioritet od zbrajanja.
3. Evaluacija izraza: nakon što je formiran AST koje određuje hijerarhiju operacija, interpretator prelazi na izvršnu fazu koje se obavlja iterativnim ili rekurzivnim postupkom. U rekurzivnom postupku interpretator analizira vrijednost svakog čvora AST-a. U iterativnom postupku koriste se algoritmi kao što je algoritma ranžirnog kolodvora.
4. Rezultat: nakon evaluacije izraza interpretator vraća krajnji rezultat, a ako izraz nije ispravan generira grešku ili poruku korisniku.

4. PRAKTIČNI RAD – PRIKAZ MATEMATIČKIH FUNKCIJA U PRAVOKUTNOM KOORDINATNOM SUSTAVU

Projekt `AlgebraicExpressionParserAndInterpreter` sastoji se od četiri podprojekta: biblioteke koja implementira parser i interpretator, biblioteke s kontrolom za prikaz pravokutnog koordinatnog sustava, same GUI aplikacije te projekta s jediničnim testovima. GUI je realiziran korištenjem biblioteke Windows Forms. U glavnu formu se upisuje matematički izraz koji se želi prikazati, parametri za kontrolu prikaza te se prikazuje koordinatni sustav s grafom izraza (Slika 3). Jediničnim testovima su se testirale funkcije i klase korištene u implementaciji parsera i interpretatora.



Slika 3. GUI dio aplikacije

4.1. Parsiranje algebarskih izraza

Glavni dio projekta AlgebraicExpressionParserAndInterpreter je podprojekt Parser. Unutar njega implementirana je javna metoda `Parse` koja predstavlja automat s konačnim brojem stanja (engl. *state machine*) i u kojoj je sadržana centralna logika projekta. Automat s konačnim brojem stanja ima dva stanja: stanje prije operatora i stanje u operatoru ili operandu. Te promjene, ovisno o operatorima i operandima imaju različita grananja.

Metoda `Parse` (Kôd 1) sastoji se od `for` petlje koja prolazi po znakovima zadanog teksta. U toj petlji nalazi se `switch-case` koji, ovisno o stanju parsera, obavlja odgovarajuće

```
public IExpression Parse(string text)
{
  // Inicijalizacija stogova operatora i izlaznog stoga...
  for (int pos = 0; pos < text.Length;)
  {
    switch (state)
    {
      case ParserState.SkippingWhiteSpacesAfterOperator:
        // Obrada zagrada i predznaka...
        break;
      case ParserState.SkippingWhiteSpacesBeforeOperator:
        // Obrada operatora...
        break;
      case ParserState.ReadingVariable:
        // Na stog će se gurnuti nova instanca varijable...
        break;
      case ParserState.ReadingConstant:
        // Na stog će se gurnuti konstanta...
        break;
      case ParserState.ReadingFunction:
        // Na stog će se gurnuti funkcija...
        break;
      case ParserState.ReadingNamedConstant:
        // Na stog će se gurnuti matematička konstanta...
        break;
    }
  }
  // Ako izraz nije ispravno zaključen:
  if (state != ParserState.SkippingWhiteSpacesBeforeOperator
  {
    throw new ParserException("...", text.Length);
  }
  // Evaluira se ostatak operatora na stogu:
  return FinalExpression();
}
```

operacije.

Kôd 1. Kostur metode `Parse`

4.2. Implementacijski detalji metode Parse

Na početku metode `Parse` definirana su dva stoga: `operators` i `output`. Ti stogovi moraju biti zasebni jer se izraz ne izračunava s lijeva na desno, nego se mora voditi računa o hijerarhiji operatora. Na stog `operators` idu operatori ovisno o njihovoj hijerarhiji, a na stog

output idu međurezultati. Na kraju, kada se parsiranje završi, stog output će sadržavati konačan rezultat.

U ugniježđenom grananju switch se, ovisno o trenutnoj poziciji u tekstu, obrađuju lijeva i desna zagrada (Kôd 2).

```
switch (text[pos])
{
  case '(':
    ++pos;
    break;
  case ')':
    throw new ParserException("Unexpected right parenthesis", pos);
```

Kôd 2. Obrada lijeve i desne zagrade

Unutar ugniježđenog switch grananja nalazi se još jedan switch u kojem se obrađuju predznaci ispred operanda (Kôd 3). Nakon toga, trenutno stanje se postavlja u stanje nakon pročitanog operatora pomoću metode GetStateAfterOperator, kojoj se prosljeđuje uneseni tekst i trenutna pozicija teksta.

```
default:
  //process plus/minus sign
  switch (text[pos])
  {
    case '-':
      operators.Push(Operator.Minus);
      ++pos;
      break;
    case '+':
         ++pos;
         break;
  }
  state = GetStateAfterOperator(text, ref pos);
  break;
}
break;
```

Kôd 3. Procesuiranje predznaka ispred operanda

Početno stanje parsera, definirano u objektu state je postavljeno na preskakanje praznih mjesta nakon operatora, SkippingWhiteSpacesAfterOperator. Ostala moguća stanja

parsera su SkippingWhiteSpacesBeforeOperator, ReadingVariable, ReadingConstant, ReadingFunction i ReadingNamedConstant.

U stanju SkippingWhiteSpacesBeforeOperator provjerava se sadrži li tekst još operatore te se oni obrađuju pozivom metode ProcessOperator, koja kao rezultat vraća novo stanje parsera (Kôd 4).

```
private ParserState ProcessOperator(string text, int pos)
{
  switch (text[pos])
  {
    case ')':
      ProcessRightParenthesis(pos);
      return ParserState.SkippingWhiteSpacesBeforeOperator;
    case '+':
      PushOperator(Operator.Addition);
      break;
    case '-':
      PushOperator(Operator.Subtraction);
      break;
    case '*':
      PushOperator(Operator.Multiplication);
      break;
    case '/':
      PushOperator(Operator.Division);
      break;
    case '^':
      PushOperator(Operator.Power);
      break;
    default:
      throw new ParserException("Invalid operator", pos);
  }
  return ParserState.SkippingWhiteSpacesAfterOperator;
}
```

Kôd 4. Metoda ProcessOperator

U slučaju stanja ReadingVariable (čitanje varijable), na stog će se gurnuti nova instanca varijable pomoću ugrađene metode Push, a stanje parsera će se postaviti u SkippingWhiteSpacesBeforeOperator (Kôd 5).

```
case ParserState.ReadingVariable:
  output.Push(new VariableX());
  state=ParserState.SkippingWhiteSpacesBeforeOperator;
  ++pos;
  break;
```

Kôd 5. Dodavanje varijable na stog

U slučaju stanja `ReadingConstant` (čitanje konstante) pozvat će se metoda `PushConstant` koja prima tekst matematičkog izraza i poziciju na kojoj vrijednost konstante počinje. Metoda `PushConstant` parsira tekst konstante i gura parsiranu vrijednost na stog (Kôd 6).

```
private void PushConstant(string text, ref int pos)
{
  int start = pos;
  int decimalSeparators = 0;
  while (pos < text.Length && (char.IsDigit(text[pos]) ||
         text[pos] == '.'))
  {
    if (text[pos] == '.')
    {
      ++decimalSeparators;
      if (decimalSeparators > 1)
      {
        throw new ParserException("Duplicate decimal...", pos);
      }
    }
    ++pos;
  }
  var value = double.Parse(text.Substring(start, pos - start),
              NumberStyles.None | NumberStyles.AllowDecimalPoint,
              CultureInfo.InvariantCulture);
  output.Push(new Constant(value));
}
```

Kôd 6. Metoda `PushConstant`

U slučaju stanja `ReadingFunction` pozvat će se metoda `PushFunction` koja prima ime matematičke funkcije i poziciju na kojoj se ime funkcije nalazi, a koja se kasnije koristi za prijavu greške. Metoda `PushFunction` razrješuje funkciju i gura ju na stog (Kôd 7).

```
private void PushFunction(string text, ref int pos)
{
  Operator fun = ResolveFunction(text, ref pos);
  operators.Push(fun);
  // Ime funkcije slijedi lijeva zagrada.
  if (text[pos] != '(')
  {
    throw new ParserException("Left parenthesis is missing...", pos);
  }
  operators.Push(Operator.LeftParenthesis);
  ++pos;
}
```

Kôd 7. Metoda `PushFunction`

U slučaju stanja `ReadingNamedConstant` pozvat će se metoda `PushNamedConstant` koja prima tekst matematičkog izraza i poziciju na kojoj započinje ime matematičke konstante, identificira vrijednost konstante i stavlja tu vrijednost na stog (Kôd 8).

```
private void PushNamedConstant(string text, ref int pos)
{
  string constantName = GetIdentifier(text, pos);
  if (namedConstantsToken.TryGetValue(constantName,
                                      out Constant constant))
  {
    pos += constantName.Length;
    output.Push(constant);
    return;
  }
  throw new ParserException("Unknown named constant", pos);
}
```

Kôd 8. Metoda `PushNamedConstant`

Privatna metoda `GetStateAfterOperator` (Kôd 9Error: Reference source not found), ovisno o trenutnom znaku u tekstu, vraća odgovarajuće stanje. U slučaju lijeve zagrade vraća stanje `SkippingWhiteSpacesAfterOperator`, u slučaju slova `x` vraća stanje `ReadingVariable`, a u slučaju znamenke vraća stanje `ReadingConstant`. Inače slijedi analiza niza slova i brojki koji mogu naići. Privatnoj metodi `GetIdentifier` prosljeđuje se tekst i trenutna poziciju unutar teksta i ona kao rezultat vraća pročitani niz znakova. Rezultat se sprema u varijablu `identifier`. Zatim se, pod uvjetom da varijabla `identifier` sadrži neki znakovni niz, provjerava slijedi li identificiranom nizu u tekstu lijeva zagrada. U tom slučaju se radi o imenu funkcije te se postavlja stanje u `ReadingFunction`. Ukoliko taj uvjet nije zadovoljen, radi se o matematičkoj konstanti te metoda `GetStateAfterOperator` vraća stanje `ReadingNamedConstant`. Ukoliko ništa od navedenog nije zadovoljeno, radi se o nedozvoljenom znaku te će biti bačena odgovarajuća iznimka.

```
private ParserState GetStateAfterOperator(string text, ref int pos)
{
  if (text[pos] == '(')
  {
    return ParserState.SkippingWhiteSpacesAfterOperator;
  }
  if (text[pos] == 'x')
  {
    return ParserState.ReadingVariable;
  }
  if (char.IsDigit(text[pos]))
  {
    return ParserState.ReadingConstant;
  }
  // Identificira niz slova i znamenki.
  string identifier = GetIdentifier(text, pos);
  if (identifier.Length > 0)
  {
    //Ako slijedi zagrada, onda to mora biti funkcija.
    if (pos + identifier.Length < text.Length &&
        text[pos + identifier.Length] == '(')
    {
      return ParserState.ReadingFunction;
    }
  return ParserState.ReadingNamedConstant;
  }
  throw new ParserException("Unexpected character", pos);
}
```

Kôd 9. Metoda `GetStateAfterOperator`

Metoda `FinalExpression` (Kôd 10) evaluira ostatak operatora na stogu. U petlji se provjerava ima li još operatora koje treba izračunati te se poziva metoda `EvaluteExpressionFromTop` koja izračunava izraz za operatore na vrhu stoga. Ta metoda prvo izvadi sve operatore jednakog ili višeg prioriteta od zadnjeg operatora, zajedno s odgovarajućim operandima, te ih gurne na lokalni stog kako bi se izračunali izrazi s lijeva na desno, kako se pojavljuju u originalnom tekstu. Potom procesuira lokalni stog i gurne rezultat na stog.

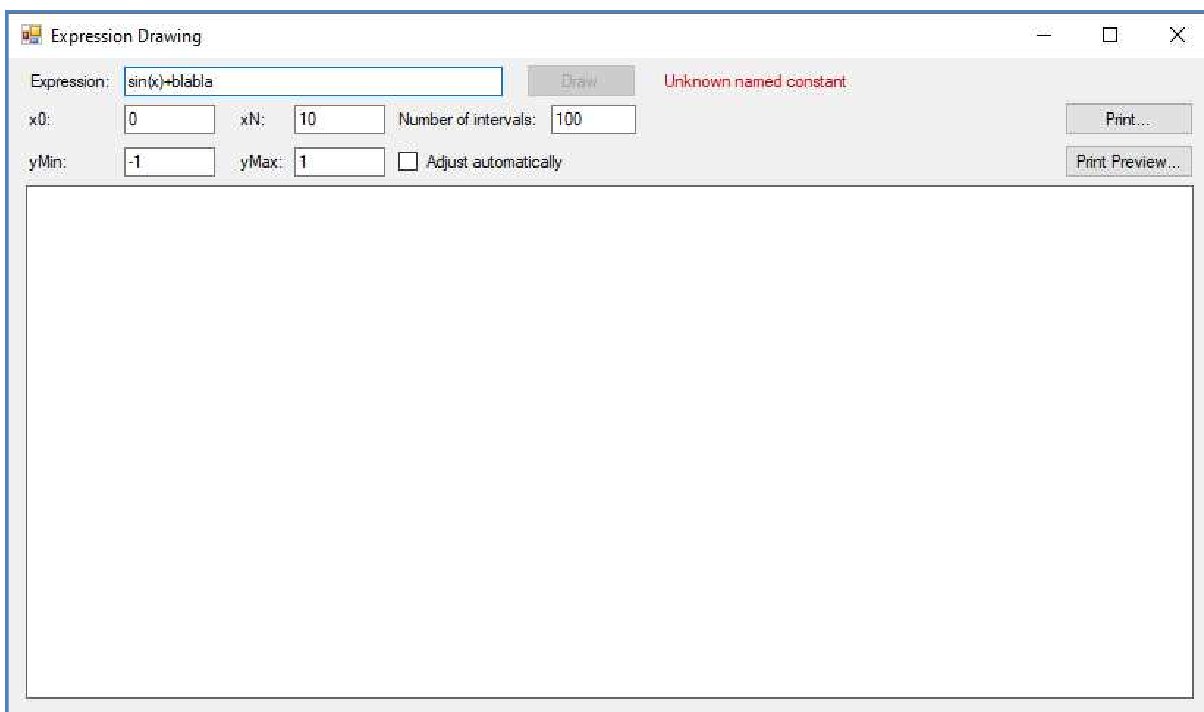
```
private IExpression FinalExpression()
{
  while (operators.Count > 0)
  {
    if (operators.Peek() == Operator.LeftParenthesis)
    {
      throw new ParserException("Right parentesis is missing...", 0);
    }
    EvaluateExpressionFromTop();
  }
  return output.Pop();
}
```

Kôd 10. Metoda `FinalExpression`

4.3. GUI dio aplikacije

GUI dio aplikacije (Slika 3) realiziran je pomoću Windows Forms okvira. Windows Forms je grafičko korisničko sučelje za izradu desktop aplikacija. U sklopu ovog projekta, za prikaz funkcije koristi se klasa `FunctionGridView` koja nasljeđuje .NET klasu `PictureBox`. Pomoću `Textbox` kontrola napravljena je mogućnost unosa teksta, u ovom slučaju matematičke funkcije, te korisnik može odabrati parametre za prikaz x-osi (*x0*, *xN*) te parametre za prikaz y-osi (*yMin*, *yMax*). Korisnik također može odabrati broj točaka u kojima se vrijednost funkcije izračunava (*Number of intervals*). Opcija *Adjust Automatically* omogućuje prikaz funkcije tako da se raspon na y-osi prilagodi unesenoj matematičkoj funkciji. Na krajnjem desnom dijelu grafičkog prikaza nalaze se tipke *Print...* i *Print Preview...*. Pritiskom na tipku *Print...* otvorit će se dijalog za odabir opcija ispisa matematičke funkcije, a na tipku *Print Preview...* otvorit će se dijalog s prikazom matematičke funkcije i njenim tekstualnim opisom.

U sklopu projekta implementirano je rukovanje pogreškama. Na primjer ako korisnik unese konstantu koja nema smisla, pojavit će se poruka o pogrešci (Slika 4). Ostale pogreške koje se pojavljuju su: 'Invalid operator', 'Right parentesis is missing or mismatched', 'Left parenthesis is missing after function name', 'Duplicate decimal separator' 'Unknown named constant' i 'Unexpected character'.



Slika 4. Prikaz pogreške u unesenom izrazu

5. ZAKLJUČAK

U okviru ovog završnog rada je napravljena aplikacija za grafički prikaz matematičkih izraza u pravokutnom koordinatnom sustavu. Problematika rada se uglavnom sastojala u implementaciji parsera, dijela koji je odgovoran za interpretaciju korisnikovog unosa. U početku se za potrebe rada implementirala funkcionalnost samo za jednostavne funkcije. Međutim, rad je zahtijevao i vođenje brige o kompleksnim izrazima koji mogu sadržavati više matematičkih funkcija, konstanti i varijabli. Bilo je potrebno analizirati i raščlaniti znak po znak cijeli izraz koji korisnik unese, uključujući lijevu i desnu zagradu, operatore množenja, dijeljenja, zbrajanja, oduzimanja i potencije, ime funkcije, ostale alfanumeričke znakove. Također je bilo potrebno implementirati rukovanje neispravnim korisnikovim unosom.

U radu je omogućen unos matematičke funkcije i generiranje grafa te funkcije u koordinatnom sustavu. Za to je bilo potrebno implementirati odgovarajući parser prema algoritmu ranžirnog kolodvora koji parsira korisnikov unos znak po znak, koristeći automat s konačnim brojem stanja te interpretator koji obrađuje konačan rezultat parsera. Algoritam ranžirnog kolodvora parsira aritmetičke i logičke izraze (ili kombinaciju oba) te se zasniva na manipulaciji stogom. Oblikovni obrazac interpretatora (engl. *interpreter*) definira način kako interpretirati neku jezičnu gramatiku ili izraz te omogućava hijerarhijsku strukturu kako bi se mogli interpretirati kompleksniji izrazi. Na temelju interpretacije rezultata bilo je moguće grafički prikazati taj rezultat koristeći biblioteku Windows Forms. Bilo je potrebno predvidjeti sva moguća stanja u automatu s konačnim brojem stanja. Isto tako bilo je potrebno ispravno grafički prikazati taj rezultat u obliku grafa u koordinatnom sustavu, što je smisao ovog projekta.

Iako je GUI kao što je Windows Forms bio dovoljan za potrebe ovog završnog rada, implementirana funkcionalnost se može unaprijediti na način da se uz korištenje bazične funkcionalnosti ovog projekta implementira napredniji GUI koji bi imao mogućnost da se na temelju korisnikovog unosa daju već gotovi prijedlozi za često korištene funkcije, da prikaz same funkcije bude veći, da uz sam korisnikov unos i grafički prikaz na temelju korisnikovog unosa postoji prikaz dodatnih osobina funkcije, na primjer, sjecišta koordinatnih osi, domena funkcije, parnost, derivacija, integral, minimum odnosno maksimum. Općenito govoreći, sam grafički prikaz mogao bi biti i vizualno privlačniji korisniku, uz korištenje boja, sjenčanja ili fontova.

Također, bazična funkcionalnost ovog rada mogla bi se primijeniti u web i u mobilnim aplikacijama te bi našla praktičnu primjenu prvenstveno u školama i fakultetima, kako bi

učenici i studenti mogli 'na dlanu' vidjeti kako izgleda funkcija pri rješavanju različitih zadataka, pripremama za ispit ili kao poticaj da rade na svom znanju.

LITERATURA

1. Gamma, Erich, et al., (2000), Design Patterns: Elements of Reusable Object-Oriented Software, Pearson, pp. 243-255
2. student.vsite.hr, Rječnik koji definira regularni izraz, https://student.vsite.hr/?q=sforums/forumpage/1389 (pristupljeno 8. 7. 2024.)
3. What is a programmer language parser, https://pgrandinetti.github.io/compilers/page/what-is-a-programming-language-parser/ (pristupljeno 8. 8. 2024.)
4. Wikipedia.org, Interpreter pattern, https://en.wikipedia.org/wiki/Interpreter_pattern (pristupljeno 18. 7. 2024.)
5. Wikipedia. org, Software design pattern, https://en.wikipedia.org/wiki/Software_design_pattern (pristupljeno 18. 7. 2024.)
6. Wikipedia.org, Shunting Yard Algorithm, https://en.wikipedia.org/wiki/Shunting_yard_algorithm# (pristupljeno 16.7.2024.)

SAŽETAK

U ovom završnom radu opisana je izrada aplikacije za grafički prikaz matematičkih izraza. Za to je bilo potrebno izgraditi pripadajući parser koji analizira korisnikov unos znak po znak i interpretator koji daje krajnji rezultat koji je kasnije bilo potrebno grafički prikazati. Grafičko sučelje za prikaz grafa realizirano je korištenjem Windows Forms kontrola.

Ključne riječi: interpretator, parser, matematički izrazi, C#, algoritam ranžirnog kolodvora

SUMMARY

This paper describes implementation of application for plotting mathematical functions in Cartesian coordinate system. To achieve that, it was necessary to build an appropriate parser which parses user input and an interpreter which interprets the result required for graphical presentation. Graphical user interface was created using Windows Forms library.

Keywords: interpreter, parser, mathematical expressions, C#, shunting yard algorithm