Sveučilište Jurja Dobrile u Puli

Fakultet informatike u Puli

Tehnički fakultet u Puli

Funkcijski znanstveni kalkulator: Izvještaj

Lucija Josipović Deranja, Marko Papić
Mentor: doc. dr. sc. Siniša Miličić

Pula, 25. lipnja, 2024.

Sadržaj

1 Uvod

Funkcijsko programiranje je paradigma koja se posljednjih godina sve više koristi u razvoju softvera zbog svojih jedinstvenih karakteristika koje omogućuju jednostavnije i efikasnije upravljanje složenim operacijama. Jedan od jezika koji se često koristi za funkcijsko programiranje je Haskell. Haskell je čist funkcijski jezik što znači da sve funkcije u Haskell-u ne mijenjaju stanje i nemaju nuspojava čime se osigurava predvidljivost i jednostavnost razumijevanja koda.

Ovaj rad istražuje prednosti funkcijskog programiranja kroz izradu znanstvenog kalkulatora u Haskell-u. Funkcijsko programiranje nudi nekoliko ključnih prednosti u usporedbi s imperativnim i objektno orijentiranim pristupima. Među najvažnijim prednostima su imutabilnost podataka, čiste funkcije, visoka modularnost i izražajnost. U svijetu gdje softverski sustavi postaju sve složeniji, sposobnost jednostavnog upravljanja i razumijevanja tih sustava postaje ključna za uspješan razvoj i održavanje softvera.

Haskell kao predstavnik čistih funkcijskih jezika omogućuje programerima da pišu kod koji je lako razumljiv, testiran i održavan. Njegova sposobnost rukovanja složenim operacijama na elegantan način čini ga idealnim za razvoj aplikacija kao što su znanstveni kalkulatori koji zahtijevaju visoku preciznost i pouzdanost.

Cilj ovog projekta je izraditi znanstveni kalkulator koristeći Haskell kako bi se demonstrirale prednosti funkcijskog programiranja u upravljanju složenim operacijama i održavanju čistoće i modularnosti koda. Konkretno, želimo:

- Prikazati kako funkcijsko programiranje olakšava upravljanje složenim operacijama - Kalkulator će uključivati različite matematičke funkcije koje će biti implementirane kao čiste funkcije, pokazujući kako se složene operacije mogu jednostavno kombinirati i proširivati.
- Diskutirati prednosti i izazove pri izgradnji kalkulatora u funkcijskom stilu - Kroz razvoj projekta identificirat ćemo ključne prednosti kao što su jednostavnost testiranja i održavanja te izazove kao što su krivulja učenja i performanse.
- Usporediti funkcijski pristup s imperativnim pristupima - Usporedba će se temeljiti na ključnim aspektima kao što su modularnost, testiranje, performanse i čitljivost koda.

Rad je podijeljen u nekoliko ključnih dijelova. Prvo, detaljna analiza kako funkcijsko programiranje olakšava upravljanje složenim operacijama. To uključuje pregled osnovnih koncepta funkcijskog programiranja kao što su imutabilnost, čiste funkcije i kombinatorna logika. Zatim diskusija prednosti i izazovi pri izgradnji kalkulatora u funkcijskom stilu uz konkretne primjere i ilustracije. Dalje, usporedba funkcijskog pristupa s imperativnim pristupima koristeći primjere iz stvarnog svijeta kako bismo ilustrirali razlike i sličnosti između ovih paradigmi. Na kraju, biti će prikazana arhitektura i dizajn kalkulatora.

Razvoj znanstvenog kalkulatora u Haskell-u nije samo teorijski rad, već ima praktične implementacije. Korištenje funkcijskih jezika poput Haskell-a može značajno smanjiti broj grešaka u kodu i olakšati održavanje velikih i složenih sustava. Očekujemo da će ovaj projekt pružiti vrijedne uvide u to kako funkcijsko programiranje može poboljšati proces razvoja softvera i potaknuti širu upotrebu funkcijskih jezika u projektima. Razvoj funk-

cijskog znanstvenog kalkulatora također će poslužiti kao koristan resurs za programere i studente koji žele naučiti više o funkcijskom programiranju i njegovim prednostima. Kroz konkretne primjere i diskusije cilj je pružiti duboko razumijevanje funkcijskog programiranja i pokazati kako se može primijeniti na stvarne projekte.

2 Analiza kako funkcijsko programiranje olakšava upravljanje složenim operacijama

Funkcijsko programiranje (FP) donosi niz prednosti koje značajno olakšavaju upravljanje složenim operacijama u razvoju softvera. Kroz ovu analizu razmotrit ćemo ključne aspekte FP-a koji doprinose ovoj jednostavnosti, uključujući koncept čistih funkcija, nemjenjivih podataka te kako FP pristup omogućava kompoziciju funkcija za složene operacije.

2.1 Čiste funkcije

Jedan od temeljnih koncepata FP-a su čiste funkcije. Čista funkcija je ona koja za iste ulazne vrijednosti uvijek vraća iste izlazne vrijednosti i nema nikakvih nuspojava. Ovo svojstvo ima nekoliko važnih implikacija na razvoj softvera:

- Predvidljivost i testabilnost: Čiste funkcije su vrlo predvidljive jer njihov izlaz ovisi isključivo o njihovim ulazima. Ovo značajno olakšava testiranje i otkrivanje grešaka jer se svaka funkcija može testirati izolirano od ostatka sustava.
- Jednostavno razmišljanje: Programeri mogu razmišljati o svakoj funkciji kao o crnoj kutiji koja obavlja specifičan zadatak. Ovo pojednostavljuje razumijevanje i održavanje koda jer je lako pratiti tok podataka kroz niz funkcija.
- Lakša paralelizacija: Budući da čiste funkcije nemaju nuspojave i ne ovise o vanjskom stanju, mogu se lako paralelizirati. Ovo je ključna prednost u modernom računarstvu gdje je iskorištavanje višejezgrenih procesora sve važnije.

U Haskell-u, čiste funkcije su standard. Na primjer:

```
add :: Int -> Int -> Int
add x y = x + y
```

Ova funkcija zbraja dva broja i uvijek će za iste ulaze vratiti isti rezultat bez ikakvih nuspojava.

```
testAdd = add 2 3 == 5
```

Čiste funkcije također omogućuju lakše razmišljanje o programima jer svaka funkcija može biti analizirana izolirano, bez brige o vanjskim utjecajima.

2.2 Nemjenjivi podaci

Drugi ključni koncept FP-a su nemjenjivi podaci. U funkcijskim jezicima, jednom kada je vrijednost dodijeljena varijabli, ona se ne može promijeniti. Ovo svojstvo donosi nekoliko važnih prednosti:

- Jednostavnije praćenje stanja: Budući da se podaci ne mijenjaju, mnogo je lakše pratiti stanje programa. Programeri ne moraju brinuti o neočekivanim promjenama podataka koje mogu dovesti do grešaka.
- Lakša paralelizacija: Kao i kod čistih funkcija, nemjenjivost podataka olakšava paralelizaciju jer ne postoji rizik od istovremenih izmjena istih podataka.

- Sigurnost i pouzdanost: Nemjenjivi podaci smanjuju mogućnost grešaka koje proizlaze iz nenamjernih promjena podataka, što povećava ukupnu sigurnost i pouzdanost programa.

U Haskell-u su sve varijable nemjenjive po defaultu. Na primjer, ako definiramo listu:

```
let numbers = [1, 2, 3, 4, 5]
```

Ova lista se ne može promijeniti, ali možemo kreirati novu listu na temelju stare:

```
originalList = [1, 2, 3]
let newNumbers = numbers ++ [6]
```

Ovakav pristup olakšava praćenje toka podataka kroz program i osigurava da se stanja ne mijenjaju na neočekivane načine.

2.3 Kompozicija funkcija

Kompozicija funkcija je još jedan moćan koncept FP-a. Omogućava stvaranje složenih operacija kombiniranjem jednostavnih funkcija. Ovo se postiže putem viših rednih funkcija (higher-order functions) koje mogu uzimati druge funkcije kao argumente ili vraćati funkcije kao rezultate.

- Reusabilnost: Kompozicija funkcija potiče reusabilnost koda. Jednom napisane funkcije mogu se koristiti kao građevni blokovi za stvaranje složenijih operacija, smanjujući potrebu za dupliciranjem koda.
- Modularnost: Kod pisan u FP stilu je vrlo modularan. Svaka funkcija obavlja specifičan zadatak i može se jednostavno kombinirati s drugim funkcijama. Ovo olakšava održavanje i proširivanje sustava.
- Deklarativnost: FP omogućava deklarativan stil programiranja, gdje programer opisuje što se treba učiniti umjesto kako to učiniti. Ovaj stil je često bliži prirodnom jeziku i može biti lakše razumljiv.

Primjer kopozicija je funkcija compose koja kombinira dvije funkcije:

```
compose :: (b -> c) -> (a -> b) -> a -> c
compose f g x = f (g x)
```

Korištenjem compose, možemo kombinirati jednostavne funkcije kako bismo kreirali složenije operacije:

```
increment :: Int -> Int
increment x = x + 1

square :: Int -> Int
square x = x * x

incrementThenSquare :: Int -> Int
incrementThenSquare = compose square increment
```

Ova kompozicijska logika omogućava jednostavno proširenje funkcionalnosti kalkulatora bez potrebe za velikim promjenama u postojećem kodu.

Kombinacija čistih funkcija, nemjenjivih podataka i kompozicije funkcija čini FP idealnim za upravljanje složenim operacijama. Ovi koncepti smanjuju složenost, olakšavaju testiranje i održavanje koda, te omogućavaju efikasnije iskorištavanje modernih računalnih resursa. U kontekstu izgradnje znanstvenog kalkulatora, ovi principi osiguravaju da kalkulator bude pouzdan, skalabilan i jednostavan za proširenje.

3 Diskusija prednosti i izazovi pri izgradnji kalkulatora u funkcijskom stilu

Izgradnja znanstvenog kalkulatora u funkcijskom stilu donosi niz prednosti, ali i specifične izazove koji zahtijevaju pažljivo razmatranje.

3.1 Prednosti

- **Deklarativni pristup**:FP omogućava deklarativni stil programiranja, gdje se programer više fokusira na opisivanje što se treba izvršiti, umjesto na detalje kako to izvesti. Ovo čini kod čišćim i lakšim za razumijevanje, jer se naglasak stavlja na funkcionalnosti, a ne na upravljanje stanjem.
- **Modularnost i ponovna upotrebljivost**: Funkcijski stil potiče modularnost koda. Svaka funkcija obavlja specifičan zadatak i može se lako ponovno upotrijebiti u različitim dijelovima aplikacije. Ovo značajno smanjuje količinu dupliciranog koda i olakšava održavanje.
- **Jednostavno testiranje**: Kao što smo ranije spomenuli, čiste funkcije su jednostavne za testiranje jer njihov izlaz ovisi isključivo o njihovim ulazima. Ovo omogućava lako pisanje jediničnih testova koji osiguravaju ispravnost funkcionalnosti.
- **Upravljanje složenim operacijama**: Kombiniranjem jednostavnih funkcija moguće je kreirati složene operacije na način koji je lako razumljiv i održiv. Ovo je posebno važno za znanstvene kalkulatore koji često moraju obraditi složene matematičke izraze.
- **Otpornost na greške**:: Zbog svoje prirode čistih funkcija, FP često vodi smanjenju grešaka u kodu. Funkcije koje nemaju nuspojave su lakše testirati i održavati, što dovodi do stabilnijeg softvera.

3.2 Izazovi

- **Krivulja učenja**: Funkcijski stil programiranja, a posebno Haskell, može imati strmu krivulju učenja za programere koji dolaze iz imperativnih ili objektno orijentiranih jezika. Koncepti kao što su monade, lijeno izvođenje i tipovi višeg reda mogu biti izazovni za početnike.
- **Performanse**: Iako funkcijsko programiranje nudi mnoge prednosti, može imati i određene performansne nedostatke, posebno u usporedbi s visoko optimiziranim imperativnim kodom. Ovo je posebno važno za aplikacije koje zahtijevaju vrhunsku izvedbu.
- **Integracija s drugim sustavima**: Funkcijski jezici poput Haskell-a mogu imati izazove pri integraciji s drugim sustavima i bibliotekama koje su napisane u imperativnim jezicima. Ovo može zahtijevati dodatne napore i prilagodbe.
- **Specifičnost zadatka**: Neki zadaci ili domeni možda nisu prirodno pogodni za funkcijski stil programiranja. Na primjer, interaktivni korisnički interfejsi ili aplikacije koje zahtijevaju puno promjenjivog stanja mogu se lakše implementirati u imperativnom stilu.

Unatoč ovim izazovima, prednosti funkcijskog pristupa često nadmašuju nedostatke, posebno za projekte koji zahtijevaju visoku modularnost, jednostavnost održavanja i pouzdanost.

4 Usporedba funkcijskog pristupa s imperativnim pristupima

Usporedba funkcijskog i imperativnog pristupa može pomoći u razumijevanju prednosti i nedostataka svakog od njih te u odabiru odgovarajuće paradigme za određene projekte.

4.1 Imperativni pristup

Imperativni pristup temelji se na sekvencijalnom izvođenju naredbi koje mijenjaju stanje programa. Glavne karakteristike imperativnog pristupa uključuju:

- Promjenjivo stanje: Programi se temelje na promjenjivim varijablama koje mijenjaju vrijednosti tijekom izvođenja.
- Kontrola toka: Programi koriste strukture kontrole toka kao što su petlje i uvjetne naredbe za upravljanje tijekom izvođenja.
- Izravna manipulacija memorijom: Programeri često izravno manipuliraju memorijom, što može povećati učinkovitost, ali i složenost koda.

4.2 Funkcijski pristup

Funkcijski pristup temelji se na evaluaciji funkcija koje ne mijenjaju stanje. Ključne karakteristike funkcijskog pristupa uključuju:

- Nemjenjivo stanje: Sve varijable su nemjenjive i svaka promjena stvara nove varijable.
- Kompozicija funkcija: Programi se temelje na kompoziciji funkcija koje obavljaju specifične zadatke.
- Apstrakcija i modularnost: Funkcijski pristup potiče visoku razinu apstrakcije i modularnosti, što olakšava ponovno korištenje koda.

4.3 Praktični primjeri usporedbe

- Funkcijski pristup:

```
    const add = (a, b) => a + b;
const multiply = (a, b) => a * b;
```

- Imperativni pristup:

```
  function add(a, b) {
  return a + b;
}

function multiply(a, b) {
  return a * b;
}
```

U ovom primjeru, razlike su minimalne, ali kako se kompleksnost povećava, razlike između pristupa postaju očitije. U konačnici, izbor između funkcijskog i imperativnog pristupa ovisi o specifičnostima projekta, timskim vještinama i zahtjevima performansi. Funkcijsko programiranje nudi prednosti u smislu čistoće koda, modularnosti i reusabilnosti,

dok imperativni pristup može biti bolji za brzinski osjetljive aplikacije i situacije gdje je kontrola nad stanjem ključna. Kombinacija oba pristupa može pružiti najbolje od oba svijeta, ovisno o konkretnom kontekstu primjene.

5 Kako naglasiti funkcijsko programiranje

5.1 Isticanje upotrebe funkcijskih jezika i tehnika kroz cijeli projekt

Funkcijsko programiranje je temelj ovog projekta, stoga je važno dosljedno koristiti funkcijske jezike i tehnike kroz cijeli proces razvoja. Korištenjem Haskell-a, čistog funkcijskog jezika, možemo demonstrirati kako funkcijski pristup može olakšati razvoj složenih aplikacija kao što je znanstveni kalkulator. Haskell omogućuje pisanje funkcija koje ne mijenjaju stanje i nemaju nuspojava, čime se osigurava predvidljivost i jednostavnost razumijevanja koda. Na primjer, osnovne operacije kalkulatora poput zbrajanja, oduzimanja, množenja i dijeljenja mogu biti implementirane kao čiste funkcije koje uzimaju ulazne vrijednosti i vraćaju rezultate bez ikakvih nuspojava:

```
add :: Int -> Int -> Int
add x y = x + y

subtract :: Int -> Int -> Int
subtract x y = x - y

multiply :: Int -> Int -> Int
multiply x y = x * y

divide :: Int -> Int -> Maybe Int
divide _ 0 = Nothing
divide x y = Just (x `div` y)
```

Kroz cijeli projekt, fokus je na korištenju funkcijskih tehnika kao što su rekursija, visoko reda funkcije i kombinatorna logika, kako bi se pokazalo kako ove tehnike olakšavaju pisanje modularnog i održivog koda.

5.2 Demonstracija efikasnosti i jednostavnosti koda zahvaljujući čistim funkcijama i nemjenjivim podacima

Jedna od ključnih prednosti funkcijskog programiranja je upotreba čistih funkcija i nemjenjivih podataka, što rezultira efikasnijim i jednostavnijim kodom. Čiste funkcije su determinističke, što znači da za iste ulazne vrijednosti uvijek vraćaju isti rezultat, bez nuspojava. Ovo olakšava testiranje i razumijevanje koda. Na primjer, možemo implementirati funkciju za izračunavanje faktora broja koristeći rekursiju, što je prirodan način rješavanja problema u funkcijskom programiranju:

```
factorial :: Int -> Int
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

Imutabilnost podataka osigurava da jednom definirane vrijednosti ne mogu biti promijenjene, što smanjuje mogućnost grešaka uzrokovanih nenamjernim promjenama stanja.

Na primjer, rad s listama u Haskell-u uključuje stvaranje novih lista umjesto promjene postojećih, što pojednostavljuje praćenje stanja:

```
let numbers = [1, 2, 3, 4, 5]
let newNumbers = map (+1) numbers  -- [2, 3, 4, 5, 6]
```

5.3 Usporedba s tradicionalnim pristupima u razvoju softvera

Tradicionalni, imperativni pristupi često se oslanjaju na promjenjivo stanje i sekvencijalno izvršavanje naredbi, što može dovesti do složenosti i teškoća u održavanju koda. Funkcijsko programiranje, s druge strane, koristi čiste funkcije i imutabilne podatke, što smanjuje složenost i olakšava razumijevanje i održavanje koda. Na primjer, u imperativnom pristupu, kalkulator bi mogao koristiti promjenjive varijable za pohranu međurezultata, što može dovesti do teškoća u praćenju toka podataka:

```
result = 0
result += 5
result *= 2
result -= 3
```

U funkcijskom pristupu, međurezultati se definiraju kao nepromjenjive vrijednosti, što omogućuje jasnije razumijevanje toka podataka:

```
calculate :: Int
calculate = let result1 = 5
            result2 = result1 * 2
            result3 = result2 - 3
        in result3
```

5.4 Analiza prednosti funkcijskog pristupa u kontekstu složenih matematičkih operacija i upravljanja stanjem

Funkcijsko programiranje nudi nekoliko ključnih prednosti u kontekstu složenih matematičkih operacija i upravljanja stanjem. Kroz korištenje čistih funkcija, možemo lako kompozirati složene operacije iz jednostavnijih funkcija, čime se olakšava razumijevanje i održavanje koda. Na primjer, izgradnja složenih matematičkih funkcija u Haskell-u može biti jednostavna i modularna zahvaljujući kombinaciji čistih funkcija:

```
calculateExpression :: Int -> Int -> Int
calculateExpression x y = add (multiply x 2) (subtract y 3)
```

Upravljanje stanjem u funkcijskom programiranju se često rješava putem monada, koje omogućuju enkapsulaciju stanja i nuspojava na čisti način. State monada je primjer kako se može upravljati stanjem bez potrebe za promjenjivim varijablama:

```
import Control.Monad.State

type CalcState = State Int
```

```
increment :: CalcState ()
increment = modify (+1)

decrement :: CalcState ()
decrement = modify (subtract 1)

runCalc :: CalcState a -> Int -> (a, Int)
runCalc = runState

main :: IO ()
main = do
  let (_, finalState) = runCalc (increment >> increment >> decrement) 0
  print finalState  -- Output will be 1
```

Ovaj pristup omogućuje jednostavno upravljanje stanjem na način koji je lako razumljiv i testiran, čime se smanjuje mogućnost grešaka i olakšava održavanje koda. Naglašavanje funkcijskog programiranja kroz upotrebu Haskell-a u ovom projektu omogućuje demonstraciju prednosti čistih funkcija, nemjenjivih podataka i modularnosti koda. Kroz usporedbu s tradicionalnim, imperativnim pristupima, jasno se vide prednosti funkcijskog pristupa, posebno u kontekstu složenih matematičkih operacija i upravljanja stanjem. Funkcijsko programiranje ne samo da pojednostavljuje razvoj i održavanje složenih aplikacija, već također pruža alat za pisanje jasnijeg, predvidljivijeg i pouzdanijeg koda.

6 Problemi s kojima smo se susreli u kodu

- **Kompilacijske greške**: Imali smo problema s kompilacijom zbog grešaka u parseru, posebno s Token.whiteSpace i setCursorPosition.
- **Interakcija s GUI-jem**: Upravljanje stanjem unutar GUI-ja i rukovanje događajima zahtijevalo je upotrebu monada, što je izazovno u funkcijskom kontekstu.
- **Evaluacija izraza**: Implementacija evaluacije složenih matematičkih izraza zahtijevala je pažljivo rukovanje različitim operacijama i prioritetima operatora.
- **Konverzija rezultata**: Prikaz rezultata u čisto decimalnom obliku bez eksponencijalne notacije bio je dodatni izazov.
- **Upravljanje zavisnostima**: Instalacija i upravljanje zavisnostima u Haskell projektu, posebno biblioteke kao što su Threepenny i Parsec, zahtijevalo je dodatnu pažnju.
- **Upravljanje zagradama**: Imali smo problema s postavljanjem korisničkog unosa unutar zagrada u parseru. Parser nije mogao ispravno obraditi izraze gdje su se zagrade koristile za grupiranje operacija, što je zahtijevalo dodatno prilagođavanje parsiranja i logike evaluacije izraza.
- **Uvoz CSS-a**:Nismo mogli uvesti CSS izvana (style.css), pa smo morali definirati sve CSS stilove unutar Main.hs. Ovo je dodalo dodatnu složenost jer smo morali ručno upravljati stilovima unutar Haskell koda, umjesto da koristimo vanjske CSS datoteke.

7 Arhitektura i dizajn kalkulatora

Arhitektura i dizajn znanstvenog kalkulatora u Haskell-u uključuju nekoliko ključnih komponenti koje omogućuju upravljanje složenim operacijama na modularan i održiv način.

7.1 Struktura programa

Struktura programa podijeljena je u nekoliko modula koji svaki obavljaju specifičan zadatak:

- **Main.hs**: Glavni modul koji upravlja korisničkim sučeljem (GUI) i interakcijama korisnika. Ovaj modul koristi biblioteku Threepenny za kreiranje GUI-ja.
- **MathOperations.hs**: Modul koji sadrži implementacije matematičkih funkcija. Sve funkcije su čiste i nemaju sporedne efekte.
- **Znanstveni-kalkulator.cabal**:modul za dodavanje zavisnosti.

7.2 Arhitektura

- **Korisničko sučelje**:
  - GUI je kreiran koristeći Threepenny biblioteku. Sadrži tekstualno polje za unos izraza, dugmad za različite matematičke operacije, i dugme za izračunavanje rezultata.
  - Korisnički unos se obrađuje i prikazuje rezultat u realnom vremenu.
- **Parser i evaluacija izraza**:
  - Implementiran je parser koristeći Parsec biblioteku za parsiranje matematičkih izraza.
  - Funkcije za različite operacije su definirane u MathOperations modu, uključujući zbrajanje, oduzimanje, množenje, dijeljenje, eksponentne funkcije, logaritme, trigonometriske funkcije, itd.
- **Logika i izračunavanje**:
  - Izraze unutar tekstualnog polja parsira parser, a zatim se evaluiraju pomoću funkcija definiranih u MathOperations modulu.

7.3 Opis Main.hs

Main.hs je glavni modul koji upravlja korisničkim sučeljem (GUI) i interakcijama korisnika pomoću Threepenny biblioteke. Ovaj modul sadrži funkcije za postavljanje korisničkog sučelja, parsiranje matematičkih izraza i prikaz rezultata.

7.3.1 Glavne funkcije i komponente:

- **Imports**:
  - Modul uvozi potrebne biblioteke za rad s GUI-jem (Threepenny), parsiranjem (Parsec) i funkcijskim operacijama.
- **Lexer i Parser**:
  - Definirane su funkcije za parsiranje matematičkih izraza koristeći Parsec biblioteku. Lexer se koristi za prepoznavanje tokena, a parser za parsiranje izraza.
  - Funkcija exprParser koristi buildExpressionParser za izgradnju parsera s tablicom operatora.

- **Funkcije za parsiranje i evaluaciju**:
  - parseExpression: Parsira ulazni string u matematički izraz koristeći zadani parser.
  - factor: Definira različite faktore (brojevi, zagrade, funkcije) koje parser prepoznaje i obrađuje.
- **Glavna funkcija main**:
  - main: Postavlja GUI pomoću Threepenny biblioteke i pokreće aplikaciju.
- **Setup funkcije za GUI**:
  - setup: Definira izgled i funkcionalnost korisničkog sučelja, uključujući tekstualno polje za unos izraza, dugmad za različite operacije i područje za prikaz rezultata.
  - Stilovi (CSS) su definirani unutar Haskell koda koristeći styleElement.
- **Postavljanje dugmadi i funkcija za unos**:
  - setup: Različita dugmad za matematičke operacije i brojeve su postavljena s odgovarajućim događajima (on UI.click).
  - Funkcije za dodavanje operatora i brojeva u tekstualno polje (appendOp, appendNum).
- **Evaluacija izraza i prikaz rezultata**:
  - setup: Kada se klikne dugme za izračunavanje, uneti izraz se parsira i evaluira. Rezultat se prikazuje u rezultatskom polju.
  - Funkcija evaluateExpression koristi parseExpression za parsiranje i evaluaciju izraza.

7.3.2 Threepenny:

Threepenny-GUI je biblioteka za Haskell koja omogućuje razvoj grafičkih korisničkih sučelja (GUI) koristeći web tehnologije. Koristi web preglednik kao prikazivač korisničkog sučelja, što znači da sučelje aplikacije može biti napisano koristeći HTML, CSS i JavaScript, dok se logika aplikacije implementira u Haskellu.

Zašto koristiti Threepenny-GUI?

- **Jednostavnost i pristupačnost**: Threepenny-GUI omogućuje jednostavan razvoj GUI aplikacija u Haskellu bez potrebe za učenjem složenih GUI frameworka kao što su Qt ili GTK.
- **Korištenje web tehnologija**: Korištenjem HTML-a, CSS-a i JavaScript-a za izradu korisničkog sučelja, Threepenny-GUI omogućuje fleksibilno i moćno dizajniranje sučelja. Web tehnologije su dobro poznate i široko korištene, što olakšava dizajn i razvoj sučelja.
- **Integracija s Haskellom**: Logika aplikacije implementirana je u Haskellu, što omogućuje korištenje prednosti funkcijskog programiranja, poput čistih funkcija i nepromjenjivih podataka. To rezultira robusnijim i održivijim kodom.

Gdje koristimo Threepenny-GUI u našem projektu?

U našem projektu, Threepenny-GUI koristimo za izradu korisničkog sučelja znanstvenog kalkulatora. Evo kako ga koristimo:

- **Izrada korisničkog sučelja**: Koristimo HTML elemente kao što su `textarea`,

`button`, i `div` za izradu sučelja kalkulatora. CSS stilovi koriste se za dizajniranje i raspored elemenata sučelja.

- **Interakcija s korisnikom**: Pomoću Threepenny-GUI događaja (eventova), kao što su `click` događaji na tipkama, korisnici mogu unositi izraze i pokretati izračune. Rezultati izračuna prikazuju se u elementima sučelja.
- **Upravljanje stanjem aplikacije**: Korištenjem Haskellovih `IORef`-ova, Threepenny-GUI nam omogućuje praćenje i ažuriranje stanja aplikacije, kao što je zadnji izračunati rezultat (`ans`).

7.3.3 Primjer koda

```
main :: IO ()
main = do
    static <- loadStatic
    ansRef <- newIORef 0.0
    startGUI defaultConfig { jsPort = Just 8023, jsStatic = Just static }
       (setup ansRef)

setup :: IORef Double -> Window -> UI ()
setup ansRef window = do
    return window # set title "Znanstveni kalkulator"

    -- HTML elementi za korisniko suelje
    input <- UI.textarea #. "input" # set (attr "placeholder") "Unesite
       izraz"
    resultLabel <- UI.span #. "result" # set text "Rezultat e biti
       prikazan ovdje"
    calculateButton <- UI.button #. "calculate-button" # set text "="

    -- Dodavanje dogaaja za tipke
    on UI.click calculateButton $ \_ -> do
        inputExpr <- get value input
        ans <- liftIO $ readIORef ansRef
        let result = evaluateExpression ans inputExpr
        case result of
            Left err -> element resultLabel # set text ("Greka: " ++ show
               err)
            Right val -> do
                element resultLabel # set text ("Rezultat: " ++ show val)
                liftIO $ writeIORef ansRef val

    -- Postavljanje elemenata u prozor
    getBody window #+
        [ element input
        , element calculateButton
        , element resultLabel
        ]
```

```
-- Number buttons
   oneButton <- UI.button #. "button-num" # set UI.text "1"
   twoButton <- UI.button #. "button-num" # set UI.text "2"
   threeButton <- UI.button #. "button-num" # set UI.text "3"
   fourButton <- UI.button #. "button-num" # set UI.text "4"
   fiveButton <- UI.button #. "button-num" # set UI.text "5"
   sixButton <- UI.button #. "button-num" # set UI.text "6"
   sevenButton <- UI.button #. "button-num" # set UI.text "7"
   eightButton <- UI.button #. "button-num" # set UI.text "8"
   nineButton <- UI.button #. "button-num" # set UI.text "9"
   zeroButton <- UI.button #. "button-num" # set UI.text "0"
   dotButton <- UI.button #. "button-num" # set UI.text "."

   -- Control buttons container
   controlButtonsContainer <- UI.div #. "control-buttons-container" #+
       [ element acButton, element backspaceButton ]

   -- Posloi elemente u GUI
   void $ getBody window #+
       [ element styleElement
       , UI.div #. "container" #+
           [ element input
           , UI.div #. "button-container" #+
               [ UI.div #. "functions-container" #+
                   [ element squareButton, element powerButton, element
                      negPowerButton
                   , element absButton,element piButton, element
                      sqrtButton
                   , element sinButton, element cosButton, element
                      tanButton
                   , element logButton, element lnButton, element eButton
                   ]
               , UI.div #. "numbers-container" #+
                   [ element nineButton, element eightButton, element
                      sevenButton, element mulButton
                   , element fourButton, element fiveButton, element
                      sixButton, element divButton
                   , element oneButton, element twoButton, element
                      threeButton, element addButton
                   , element zeroButton, element dotButton, element
                      ansButton, element subButton
                   , element lparenButton, element rparenButton, element
                      percentButton
                   ]
               ]
           , element controlButtonsContainer
           , element calculateButton
           , element resultLabel]]
```

7.4 Opis MathOperations.hs

MathOperations.hs sadrži implementacije matematičkih funkcija koje se koriste u kalkulatoru. Sve funkcije su implementirane kao čiste funkcije bez sporednih efekata.

7.4.1 Glavne funkcije:

- **Imports**:
  - Modul uvozi potrebne biblioteke za rad s matematičkim operacijama.
- **Definicije matematičkih funkcija**:
  - `add`: Funkcija za zbrajanje dva broja.
  - `subtract'`: Funkcija za oduzimanje dva broja.
  - `multiply`: Funkcija za množenje dva broja.
  - `divide`: Funkcija za dijeljenje dva broja, vraća Either tip koji omogućava rukovanje greškama (dijeljenje s nulom).
  - `power`: Funkcija za eksponenciranje.
  - `square`: Funkcija za kvadriranje broja.
  - `logarithm`: Funkcija za logaritam s proizvoljnom bazom, vraća Either tip za rukovanje greškama.
  - `sin'`, `cos'`, `tan'`: Trigonometrijske funkcije (sinus, kosinus, tangens).
  - `sqrt'`: Funkcija za kvadratni koren, vraća Either tip za rukovanje greškama.
  - `pi'`: Konstanta π.
  - `absolute`: Funkcija za apsolutnu vrednost broja.
  - `e'`: Konstanta e.
  - `ln`: Funkcija za prirodni logaritam.
  - `percentageOf`: Funkcija za procentualni račun.
  - `negPower`: Funkcija za negativni eksponent.

7.4.2 Opis funkcija:

- **Zbrajanje, oduzimanje, množenje i dijeljenje**:
  - Implementirane kao jednostavne funkcije koje uzimaju dva broja i vraćaju rezultat.
  - `divide` vraća `Either` tip kako bi rukovao greškama (dijeljenje s nulom).
- **Eksponentne funkcije i logaritmi**:
  - `power` računa eksponent jednog broja na drugi.
  - `logarithm` računa logaritam s proizvoljnom bazom, vraća `Either` tip za greške (negativni brojevi ili baza 1).
- **Trigonometrijske funkcije**:
  - `sin'`, `cos'`, `tan'` vraćaju rezultat za dati radijan.
- **Specijalne funkcije**:
  - `sqrt'` vraća kvadratni koren broja, s rukovanjem greškama.
  - `absolute` vraća apsolutnu vrednost broja.
  - `negPower` računa negativni eksponent.

Kombinacija Main.hs i MathOperations.hs pruža potpunu implementaciju znanstvenog kalkulatora u funkcijskom stilu. GUI je upravljan pomoću Threepenny biblioteke, dok su matematičke operacije implementirane kao čiste funkcije u MathOperations modulu. Kod je modularan, lako proširiv i demonstrira ključne prednosti funkcijskog programiranja kao što su imutabilnost i čiste funkcije.

8 Operatori i izrazi

- Prefix
  - Prefix operatori su oni koji se pojavljuju ispred svojih operanda. U Haskell-u, prefiksni operatori se često koriste za funkcije koje prihvaćaju jedan operand.
  - Primjer:

```
negate x  -- primjena prefiksnog operatora negate na x
```

- Infix
  - Infix operatori su oni koji se pojavljuju između svojih operanda. U Haskell-u, operatori poput +, -, *, ==, itd., su infiksni operatori.
  - Primjer:

```
x + y  -- primjena infiksnog operatora + na x i y
```

- Token.reservedOp lexer
  - Ovaj izraz dolazi iz Parsec biblioteke i koristi se za parsiranje rezerviranih operatora (poput +, -, *, itd.) u jeziku. lexer je konfiguracija za lexer koju definirate.
  - Primjer:

```
Token.reservedOp lexer "+"  -- prepoznaje i parsira simbol "+"
```

- `>>`
  - Operator `>>` je monadski operator koji sekvencijalno izvršava dvije monadske akcije, ignorirajući rezultat prve akcije.
  - Primjer:

```
action1 >> action2
-- izvršava action1, zatim action2, vraćajući rezultat action2
```

- `id .`
  - Operator kompozicije funkcija (`.`) uzima dvije funkcije i vraća njihovu kompoziciju, koja je nova funkcija. `id` je identitetska funkcija koja vraća svoj argument nepromijenjen.
  - Primjer:

```
(id . negate) x
-- primjena negate na x,
zatim identitetska funkcija, što je isto kao i negate x
```

- `<$>`
  - Operator `fmap` (`<$>`) koristi se za primjenu funkcije na vrijednost unutar funktora.
  - Primjer:

```
negate <$> Just 5
-- primjenjuje negate na Just 5, rezultirajući Just (-5)
```

- `parens`
  - Funkcija `parens` iz Parsec biblioteke koristi se za parsiranje izraza unutar zagrada.
  - Primjer:

```
parens exprParser  -- parsira izraz unutar zagrada
```

- `<|>`
  - Operator `<|>` (`alt`) koristi se za kombiniranje dva parsera u Parsec biblioteci. Pokušava prvo parsiranje s lijeve strane, a ako ne uspije, pokušava s desne strane.
  - Primjer:

```
integer <|> float
-- pokušava parsirati integer,
a ako ne uspije, pokušava parsirati float
```

- `->`
  - Operator `->` koristi se za definiranje tipova funkcija i izraza.
  - Primjer:

```
add :: Double -> Double -> Double
-- funkcija koja uzima dva Double i vraća Double
```

- `<-`
  - Operator `<-` koristi se unutar monadskih izraza (do-notacije) za ekstrakciju vrijednosti iz monadskog konteksta.
  - Primjer:

```
do
  x <- Just 5
  return (x + 1)
  -- izvlači 5 iz Just i koristi ga za izračunavanje
```

- `::`
  - Operator `::` koristi se za anotaciju tipa.
  - Primjer:

```
x :: Double  -- x je tipa Double
```

- `Double -> Double`
  - Ovo je tip funkcije koja uzima jedan argument tipa Double i vraća vrijednost tipa Double.
  - Primjer:

```
negate :: Double -> Double
-- funkcija koja uzima Double i vraća Double
```

- `Double -> Either String Double`
  - Ovo je tip funkcije koja uzima jedan argument tipa Double i vraća vrijednost tipa Either String Double. Ovo se često koristi za funkcije koje mogu vratiti grešku.
  - Primjer:

```
sqrt' :: Double -> Either String Double
-- funkcija koja vraća ili grešku ili kvadratni koren
```

- `Double -> Double -> Either String Double`
  - Ovo je tip funkcije koja uzima dva argumenta tipa Double i vraća vrijednost

tipa Either String Double.
  - Primjer:
```
divide :: Double -> Double -> Either String Double
-- funkcija za dijeljenje koja može vratiti grešku
```

- `_ <-`
  - Ovo se koristi unutar monadske do-notacije za ignoriranje rezultata monadske akcije.
  - Primjer:
```
do
  _ <- putStrLn "Hello"
  return 5  -- ignorira rezultat putStrLn
```

- `UI.button #.`
  - Ovaj izraz koristi se za stvaranje i stiliziranje gumba u Threepenny GUI biblioteci. Operator `#.` koristi se za primjenu CSS klase na element.
  - Primjer:
```
button <- UI.button #. "btn-class"
-- stvara gumb sa CSS klasom "btn-class"
```

- `UI.div #.`
  - Slično kao `UI.button #.`, koristi se za stvaranje i stiliziranje div elementa u Threepenny GUI biblioteci.
  - Primjer:
```
div <- UI.div #. "div-class"
-- stvara div sa CSS klasom "div-class"
```

- `$`
  - Operator aplikacije funkcije (`$`) koristi se za izbjegavanje zagrada. Omogućava primjenu funkcije na argument s nižim prioritetom.
  - Primjer:
```
print $ 1 + 2  -- isto kao print (1 + 2)
```

- `_ ->`
  - Lambda izraz koji ignorira svoj argument. Često se koristi za definiranje funkcija koje ne koriste svoje argumente.
  - Primjer:
```
\_ -> "Hello"  -- lambda funkcija koja uvijek vraća "Hello"
```

- Prefix
  - Prefix operatori su oni koji se pojavljuju ispred svojih operanda. U Haskell-u, prefiksni operatori se često koriste za funkcije koje prihvaćaju jedan operand.
  - Primjer:
```
negate x  -- primjena prefiksnog operatora negate na x
```

- Infix
  - Infix operatori su oni koji se pojavljuju između svojih operanda. U Haskell-u,

operatori poput +, -, *, ==, itd., su infiksni operatori.
  - Primjer:
```
x + y  -- primjena infiksnog operatora + na x i y
```

- Token.reservedOp lexer
  - Ovaj izraz dolazi iz Parsec biblioteke i koristi se za parsiranje rezerviranih operatora (poput +, -, *, itd.) u jeziku. lexer je konfiguracija za lexer koju definirate.
  - Primjer:
```
Token.reservedOp lexer "+"  -- prepoznaje i parsira simbol "+"
```

- `»`
  - Operator `»` je monadski operator koji sekvencijalno izvršava dvije monadske akcije, ignorirajući rezultat prve akcije.
  - Primjer:
```
action1 >> action2
-- izvršava action1, zatim action2, vraćajući rezultat action2
```

- `id .`
  - Operator kompozicije funkcija (`.`) uzima dvije funkcije i vraća njihovu kompoziciju, koja je nova funkcija. `id` je identitetska funkcija koja vraća svoj argument nepromijenjen.
  - Primjer:
```
(id . negate) x
-- primjena negate na x, zatim identitetska funkcija,
što je isto kao i negate x
```

- `<$>`
  - Operator `fmap` (`<$>`) koristi se za primjenu funkcije na vrijednost unutar funktora.
  - Primjer:
```
negate <$> Just 5
-- primjenjuje negate na Just 5, rezultirajući Just (-5)
```

- `parens`
  - Funkcija `parens` iz Parsec biblioteke koristi se za parsiranje izraza unutar zagrada.
  - Primjer:
```
parens exprParser  -- parsira izraz unutar zagrada
```

- `<|>`
  - Operator `<|>` (`alt`) koristi se za kombiniranje dva parsera u Parsec biblioteci. Pokušava prvo parsiranje s lijeve strane, a ako ne uspije, pokušava s desne strane.
  - Primjer:
```
integer <|> float
-- pokušava parsirati integer,
a ako ne uspije, pokušava parsirati float
```

- `->`
  - Operator `->` koristi se za definiranje tipova funkcija i izraza.
  - Primjer:
```
add :: Double -> Double -> Double
-- funkcija koja uzima dva Double i vraća Double
```

- `<-`
  - Operator `<-` koristi se unutar monadskih izraza (do-notacije) za ekstrakciju vrijednosti iz monadskog konteksta.
  - Primjer:
```
do
  x <- Just 5
  return (x + 1)
  -- izvlači 5 iz Just i koristi ga za izračunavanje
```

- `::`
  - Operator `::` koristi se za anotaciju tipa.
  - Primjer:
```
x :: Double  -- x je tipa Double
```

- `Double -> Double`
  - Ovo je tip funkcije koja uzima jedan argument tipa Double i vraća vrijednost tipa Double.
  - Primjer:
```
negate :: Double -> Double
-- funkcija koja uzima Double i vraća Double
```

- `Double -> Either String Double`
  - Ovo je tip funkcije koja uzima jedan argument tipa Double i vraća vrijednost tipa Either String Double. Ovo se često koristi za funkcije koje mogu vratiti grešku.
  - Primjer:
```
sqrt' :: Double -> Either String Double
-- funkcija koja vraća ili grešku ili kvadratni koren
```

- `Double -> Double -> Either String Double`
  - Ovo je tip funkcije koja uzima dva argumenta tipa Double i vraća vrijednost tipa Either String Double.
  - Primjer:
```
divide :: Double -> Double -> Either String Double
-- funkcija za dijeljenje koja može vratiti grešku
```

- `_ <-`
  - Ovo se koristi unutar monadske do-notacije za ignoriranje rezultata monadske akcije.
  - Primjer:
```
do
```

```
    _ <- putStrLn "Hello"
    return 5  -- ignorira rezultat putStrLn
```

- `UI.button #.`
  - Ovaj izraz koristi se za stvaranje i stiliziranje gumba u Threepenny GUI biblioteci. Operator `#.` koristi se za primjenu CSS klase na element.
  - Primjer:

```
button <- UI.button #. "btn-class"
-- stvara gumb sa CSS klasom "btn-class"
```

- `UI.div #.`
  - Slično kao `UI.button #.`, koristi se za stvaranje i stiliziranje div elementa u Threepenny GUI biblioteci.
  - Primjer:

```
div <- UI.div #. "div-class"
-- stvara div sa CSS klasom "div-class"
```

- `$`
  - Operator aplikacije funkcije (`$`) koristi se za izbjegavanje zagrada. Omogućava primjenu funkcije na argument s nižim prioritetom.
  - Primjer:

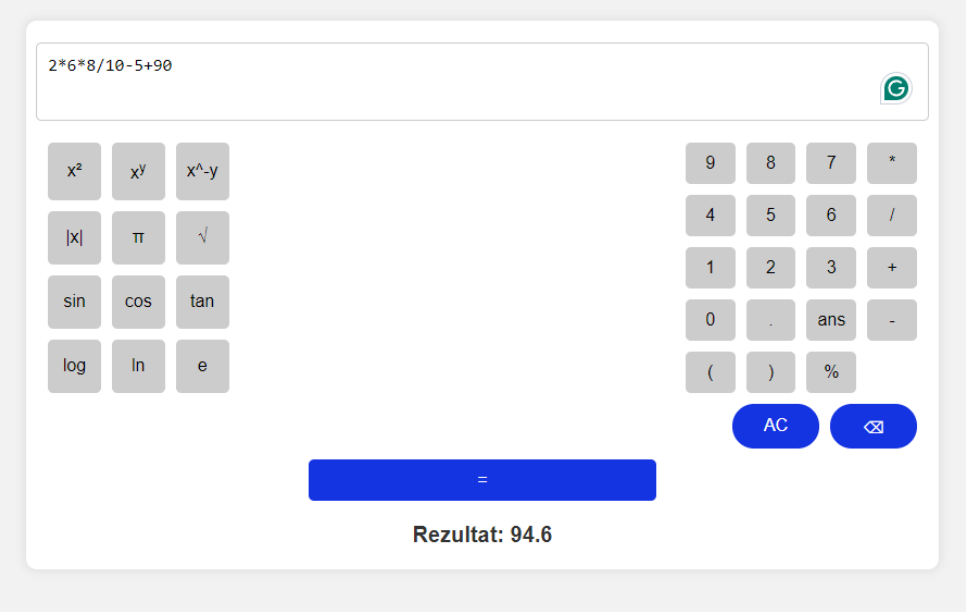
```
print $ 1 + 2  -- isto kao print (1 + 2)
```

- `_ ->`
  - Lambda izraz koji ignorira svoj argument. Često se koristi za definiranje funkcija koje ne koriste svoje argumente.
  - Primjer:

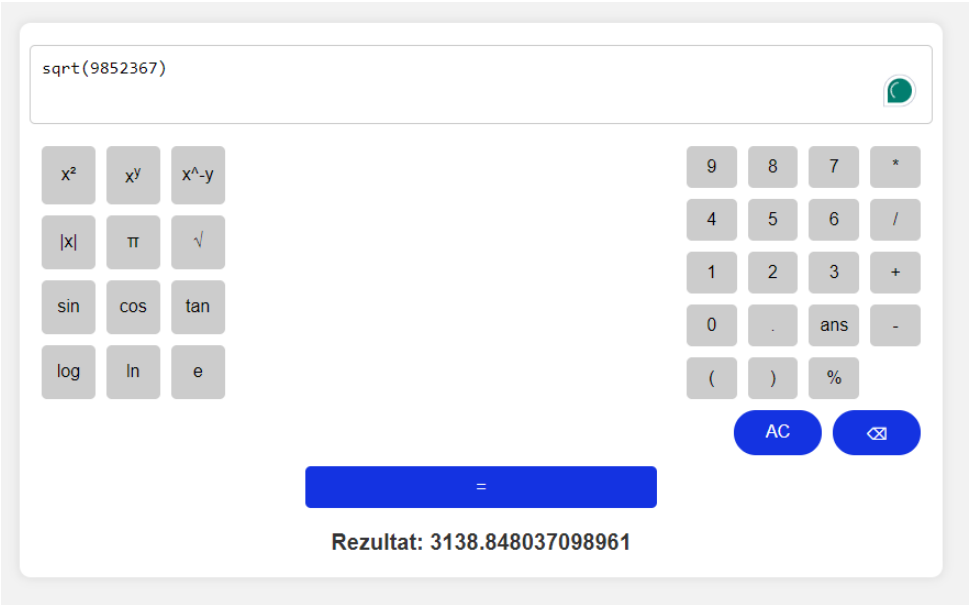
```
\_ -> "Hello"  -- lambda funkcija koja uvijek vraća "Hello"
```

9 Prikaz sučelja

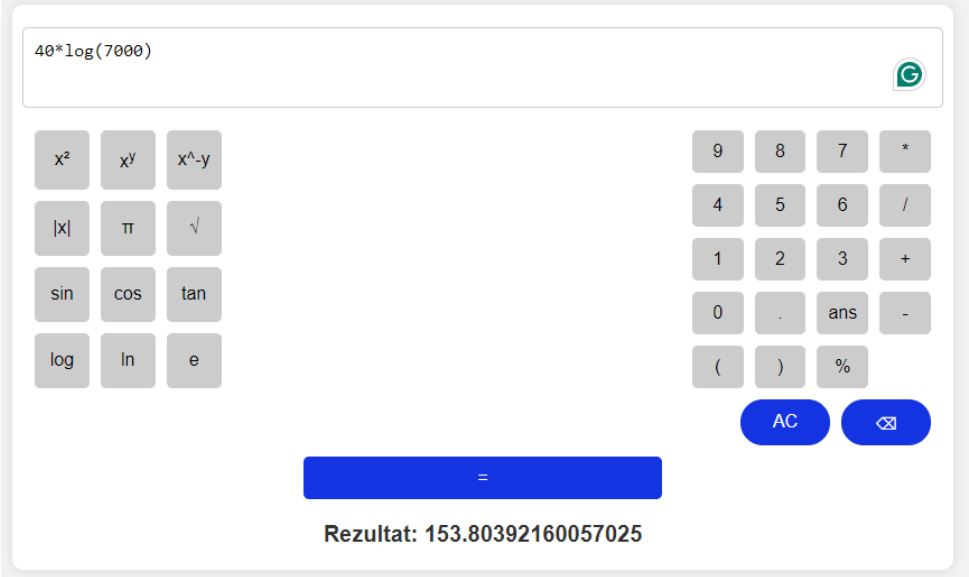
Slika 1: Prikaz početnog sučelja



Slika 2: Prikaz operacija



Slika 3: Prikaz korjenovanja



Slika 4: Prikaz množenja s logartimom

10 Zaključak

Funkcijski znanstveni kalkulator u Haskell-u predstavlja koristan primjer kako funkcijsko programiranje može olakšati upravljanje složenim operacijama i održavanje koda. Kroz ovaj projekt demonstrirane su ključne prednosti funkcijskog pristupa, uključujući modularnost, testabilnost i predvidljivost.

Unatoč određenim izazovima, kao što su krivulja učenja i performanse, funkcijski pristup nudi brojne prednosti koje ga čine vrijednim razmatranja za mnoge projekte. Funkcijski programi su često lakši za razumijevanje i održavanje, što je ključno za dugoročnu održivost softverskih projekata. Korištenje Haskell-a omogućilo je pisanje čistog, čitljivog i pouzdanog koda koji može poslužiti kao temelj za daljnji razvoj.

Ovaj projekt također naglašava važnost izbora odgovarajuće paradigme za određeni problem. Dok imperativni pristup može biti učinkovitiji u nekim slučajevima, funkcijski pristup nudi jedinstvene prednosti koje mogu značajno unaprijediti proces razvoja softvera. U budućnosti, očekujemo da će funkcijsko programiranje i dalje rasti u popularnosti kako programeri postaju sve svjesniji njegovih prednosti.

Razvoj znanstvenog kalkulatora u Haskell-u pokazuje kako se složeni matematički problemi mogu elegantno riješiti korištenjem funkcijskog pristupa. Kroz ovaj projekt, nadamo se da smo pružili vrijedne uvide u to kako funkcijsko programiranje može poboljšati razvoj softvera i potaknuli širu upotrebu funkcijskih jezika u stvarnim projektima.