SVEUČILIŠTE JURJA DOBRILE U PULI
FAKULTET INFORMATIKE U PULI

Funkcijski znanstveni kalkulator

Izradili: Andrej Korica, Dominik Ružić, Krešimir Špehar
Kolegij: Funkcijsko programiranje
Mentor: doc. dr. sc. Siniša Miličić

Pula, 2024.

Sadržaj

1. Uvod

U ovom istraživačkom radu predstavljen je znanstveni funkcijski kalkulator implementiran u programskom jeziku Haskell. Cilj projekta je demonstrirati snagu funkcijskog programiranja kroz razvoj složenog kalkulatora s grafičkim sučeljem koji omogućuje korisnicima jednostavan unos i prikaz rezultata. Kalkulator podržava razne matematičke operacije poput osnovnih aritmetičkih operacija, eksponecijacija i logaritama.

Projekt se ističe sljedećim značajkama:

Grafičko korisničko sučelje koje je intuitivno i lako za korištenje, omogućuje korisnicima unos i prikaz rezultata na jednostavan način. Kalkulator podržava složene matematičke izraze s više operatora i funkcija, omogućujući korisnicima rad s kompleksnim izrazima. Funkcijsko programiranje koristi čiste funkcije, funkcije višeg reda i obrasce podudaranja za ostvarivanje robusnog, modularnog i održivog koda. Implementacija konverzije izraza i procjene rezultata omogućuje precizno i učinkovito izračunavanje rezultata matematičkih operacija.

Korištenje imutabilnih podataka i referencijalne transparentnosti osigurava predvidljivo ponašanje programa, čineći ga lakšim za razumijevanje i održavanje. Opisani projekt pokazuje kako funkcijsko programiranje može pružiti čisto i održivo rješenje za složene probleme te ističe prednosti koje donosi ovaj pristup kod razvoja softvera.

2. Programski jezik Haskell

Haskell je funkcijski programski jezik koji je poznat po svojoj “čistoći”, izražajnosti i sparivanju uzoraka. Njegova glavna prednost leži u mogućnosti pisanja koda koji je lak za razumijevanje, održavanje i testiranje. Kao čisto funkcijski jezik, Haskell promovira nepromjenjivost podataka i referencijalnu transparentnost što rezultira predvidljivim i sigurnijim softverom.

Korištenje Haskella u razvoju navedenog kalkulatora omogućilo je iskorištavanje funkcijskih paradigmi za stvaranje robusne i modularne aplikacije. Haskellov jak sustav tipova sprječava mnoge uobičajene pogreške u fazi izgradnje dok njegove napredne značajke poput monada omogućuju elegantno upravljanje nuspojavama.

Ključne značajke Haskella:

- **Čisto funkcijsko programiranje**: U Haskellu su funkcije prvoklasni entiteti, što znači da se mogu koristiti na isti način kao i bilo koji drugi tip podataka. Mogu biti dodijeljene varijablama, proslijeđene kao argumenti funkcijama, i vraćene kao povratne vrijednosti. Čistoća također znači da su funkcije bez nuspojava, što pojednostavljuje razumijevanje i testiranje koda.
- **Nepromjenjivost podataka**: U Haskellu su podaci nepromjenjivi (eng. immutable), što znači da se vrijednosti ne mogu mijenjati nakon što su stvorene. To vodi ka većoj sigurnosti i predvidljivosti koda jer eliminira mnoge vrste pogrešaka povezanih s promjenjivim stanjima.
- **Referencijalna transparentnost**: Haskell osigurava referencijalnu transparentnost, što znači da se funkcija s istim ulazima uvijek vraća na iste izlaze bez nuspojava. Ovo svojstvo omogućava lakše razumijevanje i dokazivanje ispravnosti programa.
- **Snažna statička tipizacija**: Haskellov sustav tipova je izrazito jak i statičan, što znači da su tipovi provjereni tijekom izgradnje. Ovo omogućuje rano otkrivanje pogrešaka i osigurava visoku razinu sigurnosti i pouzdanosti koda.
- **Lijena evaluacija**: Haskell koristi lijenu evaluaciju (lazy evaluation), što znači da se izrazi ne izračunavaju dok rezultati nisu potrebni. Ova značajka omogućuje definiranje beskonačnih struktura podataka i poboljšava performanse programa odgađanjem nepotrebnih izračuna.
- **Funkcije višeg reda**: Haskell omogućuje rad s funkcijama višeg reda, što znači da funkcije mogu uzimati druge funkcije kao argumente ili ih vraćati kao rezultate. Ovo omogućava visok stupanj modularnosti i ponovne upotrebljivosti koda.
- **Monade**: Monade su napredna značajka Haskella koja omogućuje elegantno upravljanje nuspojavama i složenim operacijama poput IO, upravljanja stanjima i rukovanja greškama. One omogućuju kombiniranje čistog funkcijskog koda s imperativnim stilom programiranja kada je to potrebno.
- **Bogata standardna biblioteka**: Haskell dolazi s bogatom standardnom bibliotekom koja pruža širok spektar funkcija za rad s različitim tipovima podataka, upravljanje datotekama, mrežno programiranje i još mnogo toga. To omogućuje razvoj složenih aplikacija bez potrebe za vanjskim bibliotekama.

Korištenjem ovih značajki, Haskell omogućuje stvaranje koda koji je ne samo ispravan i učinkovit, nego i lako razumljiv i održiv, čineći ga idealnim izborom za razvoj složenih matematičkih i znanstvenih aplikacija.

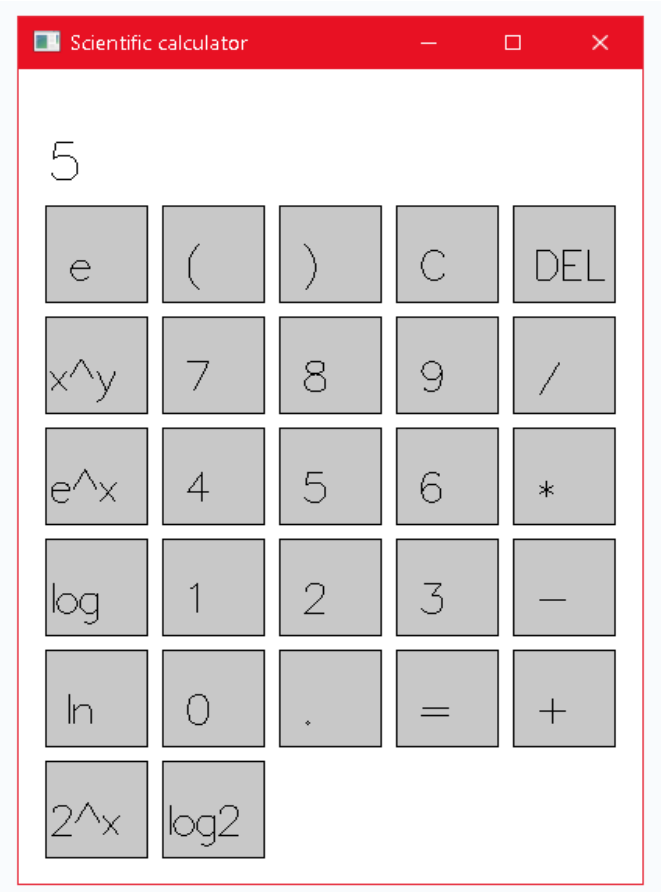
3. Upute za korištenje aplikacije

Za pokretanje kalkulatora, potrebno je pokrenuti glavnu datoteku Main.hs. Tijekom pokretanja aplikacija sve potrebne Haskell biblioteke će biti instalirane. Pokretanje aplikacije se vrši naredbom: **stack run**

3.1. Korištenje grafičkog sučelja

3.1.1. Prikaz zaslona

Nakon pokretanja, otvorit će se prozor s kalkulatorom kao na slici 1. Gornji dio prozora prikazuje trenutni izraz koji se unosi, dok se donji dio sastoji od tipki koji se može pritisnuti za unos različitih operacija i brojeva.



Slika 1: GUI kalkulatora

3.2. Tipke kalkulatora

Brojevi (0-9): Pritisnite ove tipke za unos brojeva.

Decimalna točka (.): Koristite ovu tipku za unos decimalnih brojeva.

Osnovne aritmetičke operacije (+, -, *, /): Ove tipke koriste se za osnovne matematičke operacije.

3.2.1. Eksponencijalne i logaritamske funkcije:

- **x^y:** Koristite za potenciranje.
- **2^x:** Eksponencijalna funkcija s bazom 2.
- **e^x:** Eksponencijalna funkcija s Eulerovim brojem e.
- **log:** Logaritamska funkcija (bazom 10).
- **log2**: Logaritamska funkcija s bazom 2.
- **ln:** Prirodni logaritam (baza e).

3.2.2. Posebne tipke:

- **C:** Očisti trenutni unos.
- **DEL:** Obriši zadnji uneseni znak.
- **=:** Izračunaj rezultat trenutnog izraza.
- **(i):** Zagrade za definiranje redoslijeda operacija.

3.2.3. Primjeri upotrebe

Osnovna aritmetika:

- Unesite izraz 5 + 3 * 2.
- Pritisnite = za izračun rezultata. Rezultat će biti 11.

Korištenje zagrada:

- Unesite izraz (5 + 3) * 2.

- Pritisnite = za izračun rezultata. Rezultat će biti 16.

Eksponencijalne funkcije:

- Unesite izraz 2^3.
- Pritisnite = za izračun rezultata. Rezultat će biti 8.

Logaritmi:

- Unesite izraz log(100).
- Pritisnite = za izračun rezultata. Rezultat će biti 2.

3.3. Rukovanje pogreškama

Ako unesete neispravan izraz, kalkulator će prikazati poruku o pogrešci umjesto rezultata. Provjerite jeste li ispravno unijeli sve brojeve i operacije te koristite li ispravno zagrade.

4. Faze dizajna aplikacije

4.1. Problem zagrada i prioriteta u kalkulatoru

Aplikacija se počela razvijati tako da su se najprije naveli problemi i rješenja za navedene probleme.

Prvi problem kojim smo se susreli kod znanstvenog kalkulatora je kako će program prepoznati koje operacije imaju prioritet kod računanja. Tu dolazi do primjene postfix notacije poznate i kao obrnuta poljska notacija (RPN - Reverse Polish Notation). Postfix je metoda zapisivanja aritmetičkih izraza bez potrebe za zagradama za označavanje prioriteta operacija. Ovo se postiže stavljanjem operatora iza operanda, što omogućuje da se izraz evaluira linearno, bez potrebe za analizom prioriteta operacija.

Implementacija infix to postfix koristi shunting yard algoritam koji je poznati algoritam za pretvaranje infixa u postfix. funkcija simSYA (simulated shunting yard algoritam) uzima listu stringova (koja predstavlja infiksni izraz) i vraća listu trojki koje predstavljaju stanje izlaznog niza, stoga operatora i trenutnog tokena. Korištenje funkcije scanl omogućuje praćenje stanja kroz svaki korak algoritma.

Prije svega je bitno navesti nekoliko pomoćnih funkcija koje se koriste u svrhu pretvorbe infixa u postfix.

Najprije se navela definicija prioritet svih operatora kao što možemo vidjeti na slici 2.

```
-- Define precedence for operators
prec :: String → Int
prec "^" = 4
prec "*" = 3
prec "/" = 3
prec "+" = 2
prec "-" = 2
prec "log" = 5
prec "ln" = 5
prec _ = 0
```

Slika 2: definicija prioriteta

Nadalje imamo definiciju asocijativnosti operatora, odnosno ova funkcija provjerava je li operator lijevo asocijativan. Operator ^ i logaritmi su desno asocijativni, dok su ostali operatori lijevo asocijativni. Sve asocijacije možemo vidjeti na slici 3.

```
-- Define associativity for operators
leftAssoc :: String → Bool
leftAssoc "^" = False
leftAssoc "log" = False
leftAssoc "ln" = False
leftAssoc _ = True
```

Slika 3:definicija asocijativnosti operatora

Onda imamo funkciju koja provjerava je li dani string operator. Provjerava osnovne aritmetičke operatore i logaritamske funkcije (log i ln) kao što vidimo na slici 4.

```haskell
-- Check if a string is an operator
isOp :: String -> Bool
isOp [t] = t `elem` "-+/*^"
isOp ('l' : 'o' : 'g' : _) = True
isOp ('l' : 'n' : _) = True
isOp _ = False
```

Slika 4: Funkcija za provjeru da li je string operator

Funkcija parseLog parsira logaritamske funkcije iz danog izraza. Ako je calcFuncStr jednak "ln", vraća osnovu prirodnog logaritma. Ako je calcFuncStr u formatu "log x" (gdje je x baza logaritma), parsira x i vraća odgovarajuću vrijednost. Navedeno se nalazi na slici 5.

```haskell
-- Parse logarithm
parseLog :: String -> (String, Double)
parseLog calcFuncStr
  | calcFuncStr == "ln" = ("ln", exp 1)
  | otherwise =
      let modVal = fromMaybe 10 (stripPrefix "log" calcFuncStr >>= readMaybe)
       in ("log", modVal)
```

Slika 5: Funkcija koja parsira logaritamske funkcije iz danog izraza

Nadalje krećemo sa implementacijom Shunting Yard Algoritma:

- scanl koristi funkciju f za iteriranje kroz listu xs i akumulira stanja (out - **trenutni postfix izraz**, st - **stog**, t - **trenutni token**).
- Dalje ide krajnji korak koji osigurava da se svi preostali operatori u stogu dodaju na postfix izraz nakon što su svi tokeni obrađeni.
- Funkcija f uzima trenutno stanje (izlaz, stog, trenutni token) i ažurira ga prema trenutnom tokenu t. Ako je t operator, pomiče operatore iz stoga na izlaz dok ne naiđe na operator nižeg prioriteta ili različite asocijativnosti.
- Ako je t otvorena zagrada, stavlja je na stog
- Ako je t zatvorena zagrada, premješta operatore iz stoga na izlaz do otvorene zagrade.
- Ako je t operand, dodaje ga na izlaz.
- testOp provjerava treba li premjestiti trenutni operator iz stoga na izlazni niz.

Ovdje je bio najveći izazov napraviti modularni dizajn koji omogućava dodavanje novih funkcija u postfix zapis.

Kako bi se dodala nova funkcija potrebno je:

- dodati prioritet novoj funkciji
- navesti da li se radi o lijevoj asocijaciji
- dodati ju u isOp funkciju
- dodati asocijaciju poput baze logaritma

4.2. Evaluacija postfixa

Nadalje, nakon dobivanja postfix zapisa treba ga evaluirati, odnosno vrjednovati ga. Ovaj korak je bio relativno jednostavan dokle god je postfix izraz bio ispravan. Najteži dio je bio smisliti jednostavan način za obraditi eventualne greške.

Funkcija evalPostfix prima listu stringova koja predstavlja postfiksni izraz i vraća rezultat evaluacije kao Either String Double, gdje Right sadrži rezultat evaluacije, a Left sadrži poruku o pogrešci.

Logika evalPostfixa glasi:

- Glavni dio funkcije koristi funkciju višeg reda foldl kako bi se iteriralo kroz elemente postfiksnog izraza i primjenjuje se funkcija eval na akumulator vrijednosti i na svaki element izraza.
  Po završetku foldl provjerava:
  - Ako je Right [rezultat] vraća rezultat
  - Ako je Right _ odnosno ako je stog prazan, to ukazuje na to da izraz nije ispravno formiran jer nije bilo dovoljno operanada za izvođenje svih operacija. Ako je na stogu više od jednog elementa, to ukazuje na to da postoji previše operanda ili premalo operacija u izrazu. Vraća grešku "Error: Invalid expression"
  - Ako je Left err vraća grešku
- Dalje imamo pomoćnu funkciju eval koja se brine za evaluaciju svakog tokena u izrazu te se eval sastoji od:

- Operacija s dva operanda (+, -, *, /, ^):
  - Provjerava da li se u stogu (Right (x : y : ys)) nalaze barem 2 operanda .
  - Izvršava operaciju i vraća novi stog
  - Posebno provjerava dijeljenje s nulom kod operacije /.
- Operacije s jednim operandom (npr. ln):
  - Provjerava je li operand pozitivan
  - Izračunava prirodni logaritam
- Logaritamske funkcije koji imaju bazu (npr. log):
  - Parsira bazu logaritma iz imena funkcije (ili koristi bazu 10 kao zadanu vrijednost).
  - Provjerava je li operand pozitivan.
  - Izračunava logaritam s danom bazom.
- Parsiranje brojeva iz stringa u double
  - Pokušava parsirati token kao broj koristeći pomoćnu funkciju readMaybe
  - Ako uspije, dodaje broj na stog
- Greške:
  - Ako eval naiđe na Left err, propagira grešku.

Dodavanje nove funkcije u eval je vrlo jednostavan postupak:

- potrebno je navesti da li se radi o funkciji koja prima jedan operator ili o funkciji kojoj je potreban mod
- treba navesti koja se operacija izvršava na toj funkciji
- ako postoji neko odstupanje da se vraća prikladna greška

4.3. Renderiranje grafičkog sučelja

Budući da postoji implementacija koja će izračunati matematički izraz, potrebno je dodati sučelje putem kojeg će biti moguće unijeti te dobiti izračun unesenog izraza. Najteži izazov ovog koraka je bio odabrati adekvatnu biblioteku koja omogućava kreiranje sučelja u Haskelu. Odlučili smo se za Gloss koji je jednostavan za korištenje te radi na verziji na kojoj smo inicijalizirali projekt.

Implementacije sučelja počinje tako da su na početku definirani podatkovni tipovi koji su navedeni na slici 6. Ovaj je korak bitan kako bi održali integritet tipova podataka.

```haskell
data CalculatorState = CalculatorState
  { displayText :: String,
    postfixExpr :: Maybe [String],
    evalResult :: Maybe Double,
    pressedButton :: Maybe String
  }
```

Slika 6: definiranje podatkovnih tipova varijabli

Funkcija render crta kalkulator i njegov zaslon. Koristi drawButton za crtanje svakog dugmeta. Na slici 7 možemo vidit implementaciju.

```haskell
-- Render function
render :: CalculatorState -> IO Picture
render calcState = return $ Pictures [calculator, displayTextPicture]
  where
    calculator =
      Pictures [drawButton (x, y) size label (pressedButton calcState == Just label) | (x, y, size, label) <- buttonData]
    displayTextPicture =
        Translate (-180) 200 $ Scale 0.25 0.25 $ Text (displayText calcState)
```

Slika 7: Funkcija renderiranje

Funkcija handleEvent obrađuje događaje miša. Kada se klikne na dugme, ažurira se stanje kalkulatora unutar IO monade kao što možemo vidjet na slici 8.

```haskell
-- Handle events
handleEvent :: Event -> CalculatorState -> IO CalculatorState
handleEvent (EventKey (MouseButton LeftButton) Down _ (x, y)) calcState = do
  let clickedButton = findClickedButton (x, y)
  return $ case clickedButton of
    Just label -> calcState {pressedButton = Just label}
    Nothing -> calcState
handleEvent (EventKey (MouseButton LeftButton) Up _ (x, y)) calcState = do
  let clickedButton = findClickedButton (x, y)
  return $ case clickedButton of
    Just label -> updateDisplay label calcState {pressedButton = Nothing}
    Nothing -> calcState {pressedButton = Nothing}
handleEvent _ calcState = return calcState
```

Slika 8: Implementacija funkcije za obradu klikova

Funkcija updateDisplay ažurira zaslon kalkulatora ovisno o pritisnutom dugmetu (slika 9).

Tipke obrađene ovom funkcijom:

- "C", kalkulator se resetira
- "DEL", uklanja se zadnji znak sa zaslona
- Specijalni Operatori ("x^y", "2^x", "e^x"), dodaje se odgovarajući operator
- "=", Evaluira izraz na zaslonu

Ova funkcija isto tako upravlja s greškama:

- Provjerava da li su posljednji i prvi znak label operatori kako bi se spriječile nevažeće sekvence operatora
- Ako je displayText prazan, prikazuje "Error: Empty expression".
- Ako tokenizacija ne uspije, prikazuje grešku.
- Ako posljednji znak trenutnog displayText i prvi znak label oba su operatori, stanje ostaje nepromijenjeno.

```haskell
-- Update display based on clicked button
updateDisplay :: String -> CalculatorState -> CalculatorState
updateDisplay label calcState
  | label == "C" = calcState {displayText = "", postfixExpr = Nothing, evalResult = Nothing}
  | label == "DEL" =
      let newText =
            if null (displayText calcState)
              then ""
              else init (displayText calcState)
      in calcState {displayText = newText}
  | label == "x^y" = calcState {displayText = displayText calcState ++ "^"}
  | label == "2^x" = calcState {displayText = displayText calcState ++ "2^"}
  | label == "e^x" = calcState {displayText = displayText calcState ++ "e^"}
  | label == "=" =
      if null (displayText calcState)
        then calcState {displayText = "Error: Empty expression", evalResult = Nothing}
        else case tokenize (displayText calcState) of
          Left errMsg -> calcState {displayText = errMsg, evalResult = Nothing}
          Right tokens ->
            let result = last $ simSYA tokens
                postfix = reverse $ fst3 result ++ snd3 result
            in case evalPostfix postfix of
                 Left errorMsg -> calcState {displayText = errorMsg, evalResult = Nothing}
                 Right resultValue -> calcState {postfixExpr = Just postfix, evalResult = Just resultValue, displayText = show resultValue}
  | not (null (displayText calcState)) && isOperator (last (displayText calcState)) && isOperator (head label) = calcState
  | otherwise = calcState {displayText = displayText calcState ++ label}

-- Check if a character is an operator
```

Slika 9: Funkcija za ažuriranje displaya kalkulatora

Nakon pokretanja tipke pokrene se isto pomoćna funkcija tokenize koja dijeli izraz u tokene.

Funkcija tokenize prihvaća string s kao ulaz i vraća ili grešku (Left String) ili listu tokena (Right [String]). Interno koristi pomoćnu funkciju tokenizeHelper.

Funkcija tokenizeHelper prihvaća tri argumenta:

1. s - preostali dio ulaznog stringa koji treba tokenizirati.
2. isFirst - Bool vrijednost koja označava da li je trenutni token prvi u nizu.
3. wasOperator - Bool vrijednost koja označava da li je prethodni token bio operator.

Logika Tokenize funkcije

1. Prazan string: Ako je ulazni string prazan, vraća se prazna lista tokena (Right []).
2. Prostor ili tab: Ako trenutni c (character) u stringu s predstavlja prazan prostor, funkcija ga preskače i nastavlja sa tokenizacijom preostalog stringa.
3. Operatori i zagrade: Ako je trenutni znak jedan od operatora ()+-*/^, funkcija ga tokenizira prema sljedećim pravilima:
    - Ako je isFirst True i trenutni znak je -, to znači da je - dio negativnog broja, pa se poziva tokenizeHelper na ostatku stringa. Ako je rezultat uspješan, vraća se negativan broj kao token.
    - Ako je wasOperator True i trenutni znak je -, također se tretira kao negativan broj nakon operatora.
    - Inače, trenutni znak se tokenizira kao običan operator i dodaje u listu tokena.
4. Zatvorena Zagrada: Zatvorene zagrade) se jednostavno tokeniziraju i dodaju u listu tokena.
5. Logaritamske Funkcije: Ako je trenutni znak l, funkcija provjerava da li je riječ o logaritamskim funkcijama log ili ln. Ako jeste, cijela funkcija se tokenizira i dodaje u listu tokena.
6. Brojevi i Decimalne Točke: Ako je trenutni znak decimalna točka, funkcija uzima decimalni broj i dodaje ga u listu tokena.
7. Konstanta e: Ako je trenutni znak e, tokenizira se kao konstanta e (vrijednost 2.71828182846) i dodaje se u listu tokena.
8. Nevažeći Token: Ako nijedno od pravila nije ispunjeno, funkcija vraća grešku Invalid token

5. Primjena prednosti funkcijskog programiranja u haskellu

5.1. Čiste funkcije

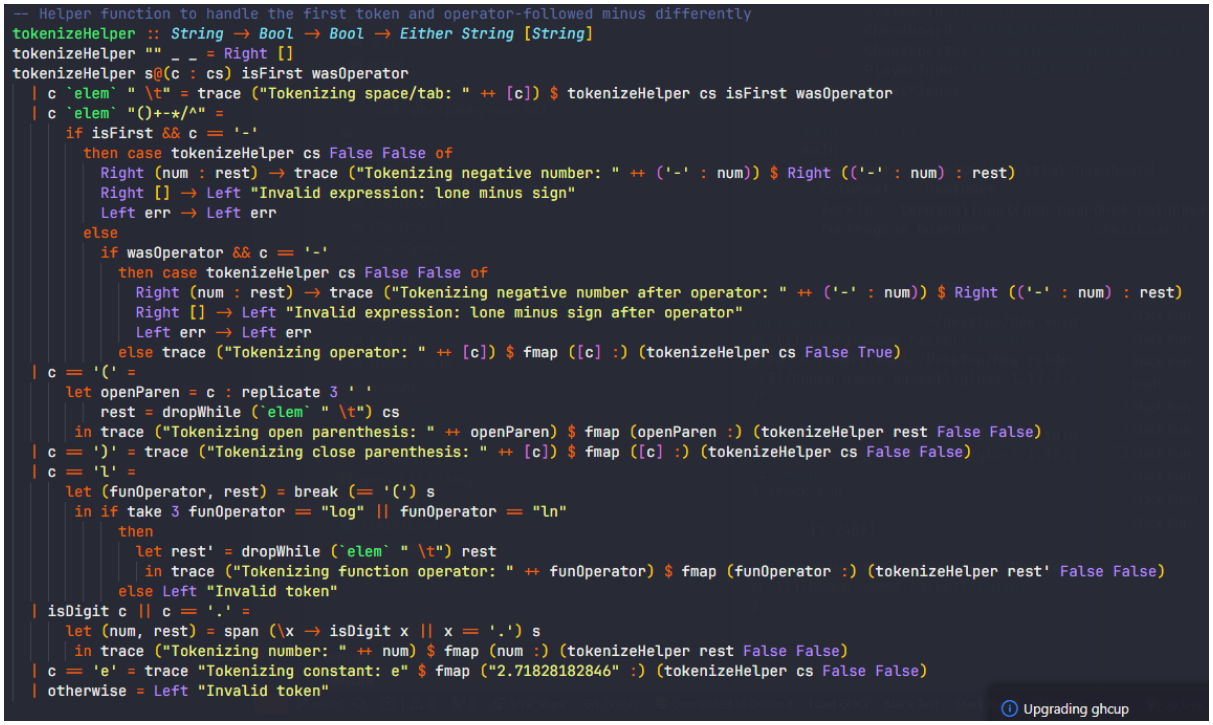
Čiste funkcije su funkcije koje za isti ulaz uvijek daju isti izlaz i nemaju nuspojave. Ovo znači da ne mijenjaju stanje programa niti ovise o vanjskom stanju. Korištenjem čistih funkcija, kod postaje predvidljiviji i lakši za testiranje jer se ponašanje funkcije može razumjeti isključivo na temelju njezinog koda. Čiste funkcije također olakšavaju razumijevanje i održavanje koda, jer nema skrivenih interakcija s drugim dijelovima programa. Primjer je funkcija na slici 10 koja uzima string i vraća rezultat koji ovisi isključivo o ulaznom stringu, bez mijenjanja stanja programa.

```haskell
-- Tokenize input string
tokenize :: String → Either String [String]
tokenize s = tokenizeHelper s True False
```

Slika 10: funkcija za tokeniziranje kao primjer za čistu funkciju

5.2. Funkcije višeg reda

Funkcije višeg reda su funkcije koje uzimaju druge funkcije kao argumente ili vraćaju funkcije kao rezultat. One omogućuju veću fleksibilnost i ponovnu upotrebljivost koda. Na primjer, funkcije poput map i filter omogućuju primjenu jedne funkcije na svaki element liste ili filtriranje elemenata liste prema nekom kriteriju. Ova sposobnost olakšava pisanje općenitih i apstraktnih rješenja koja se mogu primijeniti na različite probleme. Ovdje se fmap koristi za primjenu funkcije na rezultat tokenizeHelper, omogućujući fleksibilnu manipulaciju povratnim vrijednostima (slika 11).



Slika 11: Usklađivanje uzoraka u tokenizeHelper funkciji

5.3. Usklađivanje uzoraka

Usklađivanje uzoraka je moćan alat u Haskell-u koji omogućuje provjeru i ekstrakciju podataka iz složenih struktura podataka na jednostavan i čitljiv način. Pomoću usklađivanja uzoraka, programeri mogu napisati kod koji je intuitivan i lako razumljiv, jer se različiti slučajevi ulaznih podataka mogu jasno razlikovati i obraditi. Ovo povećava čitljivost koda i smanjuje mogućnost grešaka. Usklađivanje uzoraka se koristi u mnogim funkcijama za provjeru i ekstrakciju podataka iz složenih struktura podataka na jednostavan način. Na primjer, u funkciji tokenizeHelper na slici 11 se različiti slučajevi ulaznih podataka jasno razlikuju i obrađuju pomoću usklađivanja uzoraka.

5.4. Nepromjenjivost

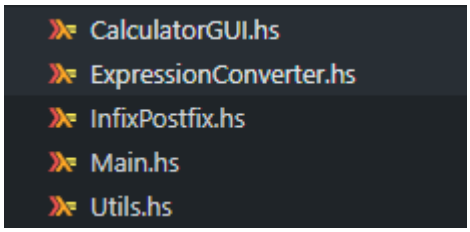
Nepromjenjivost znači da se jednom stvoreni podaci ne mogu mijenjati. Umjesto toga, kreiraju se novi podaci na temelju postojećih. Ovo osigurava dosljednost i pouzdanost podataka, jer nema neočekivanih promjena stanja koje mogu dovesti do grešaka. Nepromjenjivost također olakšava razumijevanje toka podataka kroz program i omogućuje lakše paralelno i konkurentno programiranje, jer nema potrebe za brigu o međusobnim utjecajima različitih dijelova koda. Na slici 12 može se uočiti kod koji radi tako da se tijekom promjena u stanju kalkulatora stvaraju nova stanja umjesto izmjene postojećih.

```haskell
data CalculatorState = CalculatorState
  { displayText :: String,
    postfixExpr :: Maybe [String],
    evalResult :: Maybe Double,
    pressedButton :: Maybe String
  }
```

Slika 12: Jasno definirani tipovi podataka koji predstavljaju nepromjenjivost

5.5. Modularnost

Modularnost se odnosi na organizaciju koda u manje, samostalne dijelove ili module, svaki s jasno definiranim funkcijama i odgovornostima. Ovo povećava čitljivost koda i olakšava održavanje, jer se promjene mogu lokalizirati unutar pojedinih modula bez utjecaja na ostatak programa. Modularni dizajn također olakšava ponovno korištenje koda i razdvajanje odgovornosti među različitim dijelovima aplikacije. Svaki modul na slici 13 ima jasno definirane funkcije i odgovornosti, čime se povećava čitljivost i održavanje koda.



Slika 13: moduli

5.6. Analogno objektno orijentirano supklasiranja

Analogno objektno orijentirano subklasiranje u Haskell-u se postiže korištenjem tipova podataka i klasa tipova. Tipovi podataka omogućuju definiranje novih struktura podataka, dok klase tipova omogućuju definiranje skupova funkcija koje mogu raditi s različitim tipovima podataka. Ovo omogućuje fleksibilnost i proširivost koda na način sličan objektno orijentiranom programiranju, omogućujući dodavanje novih funkcionalnosti bez mijenjanja postojećeg koda. Korištenje tipova podataka i funkcija kao što su prec, leftAssoc i isOp omogućuje fleksibilnost i proširivost koda (slika 14).

```haskell
-- Define associativity for operators
leftAssoc :: String → Bool
leftAssoc "^" = False
leftAssoc "log" = False
leftAssoc "ln" = False
leftAssoc _ = True

-- Check if a string is an operator
isOp :: String → Bool
isOp [t] = t `elem` "-+/*^"
isOp ('l' : 'o' : 'g' : _) = True
isOp ('l' : 'n' : _) = True
isOp _ = False
```

Slika 14: Analogno OO supklasiranje kod funkcija prec, leftAssoc i isOp

6. Zaključak

Kroz razvoj znanstvenog funkcijskog kalkulatora u Haskellu, prikazane su značajne prednosti i mogućnosti funkcijskog programiranja. Implementacija složenog kalkulatora s grafičkim sučeljem, koji podržava različite matematičke operacije, demonstrirala je kako Haskell može biti moćan alat za razvoj aplikacija koje zahtijevaju preciznost, modularnost i čitljivost koda. U konačnici, ovaj projekt jasno demonstrira snagu funkcijskog programiranja u razvoju složenih aplikacija i pruža dobar temelj za budući razvoj funkcionalnih softverskih rješenja. Haskell se pokazao kao pouzdan i učinkovit jezik za implementaciju matematičkih alata, zahvaljujući svojim funkcionalnim paradigmama koje podržavaju jasnoću, modularnost i sigurnost koda.

7. Literatura

Hutton, G. (2016). *Programming in haskell*. Cambridge University Press.

Thompson, S. (1996). *Haskell: The craft of Functional Programming.*

Hudak, Paul (1989). *Conception, evolution, and application of functional programming languages.*

Code, R. (2024, February 1). Parsing/Shunting-yard algorithm - Rosetta Code. Rosetta Code. https://rosettacode.org/wiki/Parsing/Shunting-yard_algorithm