Kalkulator klasične logike sudova

Martinović, Petar

SVEUČILIŠTE U ZAGREBU
FAKULTET ORGANIZACIJE I INFORMATIKE
VARAŽDIN

Petar Martinović

KALKULATOR KLASIČNE LOGIKE SUDOVA

ZAVRŠNI RAD

Varaždin, 2023.

SVEUČILIŠTE U ZAGREBU

FAKULTET ORGANIZACIJE I INFORMATIKE

V A R A Ž D I N

Petar Martinović

Matični broj: 0016143568

Studij: Informacijski sustavi

KALKULATOR KLASIČNE LOGIKE SUDOVA

ZAVRŠNI RAD

Mentor :

Doc. dr. sc. Marcel Maretić

Varaždin, rujan 2023.

Petar Martinović

Izjava o izvornosti

Izjavljujem da je moj završni rad izvorni rezultat mojeg rada te da se u izradi istoga nisam koristio drugim izvorima osim onima koji su u njemu navedeni. Za izradu rada su korištene etički prikladne i prihvatljive metode i tehnike rada.

Autor potvrdio prihvaćanjem odredbi u sustavu FOI-radovi

Sažetak

U ovom radu bit će izrađen kalkulator za ovaj formalni sistem logike. Klasična logika sudova je grana formalne logike koja se bavi proučavanjem iskaza. U klasičnoj logici sudova, iskazi su složeni od logičkih veznika poput negacije, konjunkcije, disjunkcije, implikacije i ekvivalencije. U radu će biti obrađeni osnovni koncepti klasične logike sudova, uključujući konjuktivnu i disjunktivnu normalnu formu, kao i korisnost logike sudova u praksi. U praktičnom dijelu rada, bit će izrađen kalkulator za klasičnu logiku sudova koristeći parsere u Python programskom jeziku. Izrada kalkulatora za klasičnu logiku sudova omogućuje brzo i precizno računanje iskaza i ispitivanje njihove istinitosti. Sintaktička analiza propozicijske formule na njezine jednostavnije dijelove povezane logičkim veznicima omogućuje provjeru istinitosti te formule u raznim interpretacijama, što može biti korisno u različitim primjenama, uključujući računalne znanosti, matematiku, filozofiju i druge znanstvene discipline. U konačnici, ovaj rad ima za cilj pružiti razumijevanje klasične logike sudova i njezine primjene kroz izradu kalkulatora koji može pomoći u različitim područjima istraživanja i primjene.

Ključne riječi: klasična logika sudova, formalna logika, iskazi, logički veznici, kalkulator, parsiranje, python

Sadržaj

1. Uvod

Svakodnevno se susrećemo s mnogim tvrdnjama koje mogu biti istinite ili lažne. Kako bismo proučavali i razumjeli takve tvrdnje, nastala je logika kao grana filozofije koja se bavi proučavanjem razmišljanja i zaključivanja. Jedna od grana logike je klasična logika sudova, koja se bavi formalizacijom propozicijske formule.

Klasična logika sudova se koristi za proučavanje propozicijske formule, koji se mogu izraziti kao tvrdnje koje su ili istinite ili lažne. U ovom formalnom sustavu, iskazi se sastoje od logičkih veznika kao što su negacija, konjunkcija, disjunkcija, implikacija i ekvivalencija. Ovi logički veznici se koriste za izgradnju složenijih propozicijskih formula, a cilj je provjeriti njihovu istinitost.

Klasična logika sudova ima primjenu u različitim područjima kao što su matematika, računalne znanosti, filozofija, statistika i druge znanstvene discipline. U ovom kontekstu, izrada kalkulatora klasične logike sudova korisna je za brzo i precizno računanje propozicijskih formula i provjeru njihove istinitosti.

U ovom radu će se opisati klasična logika sudova, njezini osnovni koncepti kao što su konjunktivna i disjunktivna normalna forma, te primjena logike sudova u praksi. Također, bit će izrađen kalkulator klasične logike sudova koristeći parsere u Python programskom jeziku. Cilj rada je pružiti razumijevanje klasične logike sudova i njezine primjene, te istaknuti važnost ovog formalnog sustava u različitim znanstvenim disciplinama.

Klasična logika sudova se često koristi u računalnim znanostima, posebno u umjetnoj inteligenciji, gdje se koristi za formalizaciju razmišljanja i zaključivanja koje se mogu implementirati u računalnim programima. Također, klasična logika sudova ima primjenu u filozofiji, gdje se koristi za analizu i razumijevanje različitih filozofskih teorija i argumenata.

2. Metode i tehnike rada

U ovom poglavlju treba opisati koje će metode i tehnike biti korištene pri razradi teme, kako su provedene istraživačke aktivnosti, koji su programski alati ili aplikacije korišteni.

U istraživanju literature korišteni su različiti izvori, uključujući knjige, online članke i druge izvore relevantne za ovu temu. Analiza podataka provedena je korištenjem programskog jezika Python uz pomoć "PyParsing" biblioteke, a za praktični dio korišteni su materijali kolegija Advanced Python Workshop koji su pružili veliku pomoć u izradi kalkulatora klasične logike sudova.

3. Klasična logika sudova

Klasični logički sudovi su temeljna dva logička suda: afirmacija, negacija. Ovi sudovi se temelje na klasičnoj logici koja je razvijena u Grčkoj i Rimu. Začetnikom logike (kao discipline) se smatra Aristotel 384–322 p.K. Aristotelovi sljedbenici su 6 Aristotelovih djela o logici sabrali u tzv. “Organon” (grč. instrument, alat, organ).[4]

Sud ili propozicija je deklarativna izjava koja ima svojstvo istinitosti – ima posve određenu, jednu i samo jednu vrijednost istinitosti: sud istinit ili lažan (neistinit).[4] Afirmacija je sud koji tvrdi da je nešto istinito. Na primjer, izjava "Fakultet organizacije i informatike se nalazi u Varaždinu." je afirmativni sud jer se fakultet nalazi u Varaždinu. Negacija je sud koji tvrdi da je nešto lažno. Na primjer, izjava "Fakultet organizacije i informatike pripada sveučilištu u Splitu." je negativni sud jer FOI ne pripada sveučilištu u Splitu. "Osoba x ide na Fakultet organizacije i informatike" Ovo je zamalo sud. Ovisi o x. Ovakva izjava postaje sud tek kad navedemo x. Istinitost ove izjave nije nedvosmislena već ovisi o x-u.[4]

Ovi logički sudovi su osnova za izvođenje zaključaka iz tvrdnji i razumijevanje njihovih odnosa. Klasični logički sudovi su važni za razvoj kritičkog razmišljanja i sposobnosti donošenja informiranih zaključaka, ali godinama logika se sve više razvijala pa se i danas široko koristi u matematici, informatici, umjetnoj inteligenciji, lingvistici...

Najjednostavnije lingvističke tekstove koji mogu prenositi informacije zovemo rečenicama, propozicijama (sudovima), frazama itd. Konstrukcija rečenice slijedi sintaktička pravila jezika bez obzira na značenje koje rečenica može imati.[1]. Unatoč tome što se rečenice sastoje od sintaktičkih elemenata, njihovo značenje može biti vrlo različito. Na primjer, rečenice "Petar čeka Petru" i "Petra čeka Petra" slijede ista sintaktička pravila, ali imaju potpuno različita značenja. Unatoč tome što se rečenice, propozicije i fraze sastoje od sintaktičkih elemenata, njihovo značenje ovisi o kontekstu u kojem se koriste.

Logika se stoga ne bavi samo time što je istinito, a što lažno, već prije ispravnošću naše argumentacije o tome što implicira što i valjanošću našeg razmišljanja.[3] Ova izjava opisuje logiku kao da nije samo određivanje koje su izjave istinite ili netočne. Umjesto toga, bavi se i načinom na koji te izjave koristimo za iznošenje argumenata i zaključaka. Drugim riječima, logika se bavi ispravnošću našeg zaključivanja i načinom na koji opravdavamo svoje zaključke na temelju dokaza i premisa koje imamo. Nije dovoljno samo podnijeti zahtjev; također moramo pružiti logičan argument koji to podupire.

Klasična logika, koja se temelji na principima klasične aristotelovske logike, zaista je osnovna osnova za klasičnu matematiku i većinu formalnih matematičkih teorija. Klasična logika koristi dvo vrijednosnu logiku u kojoj tvrdnje mogu biti točne (istinite) ili netočne (lažne). Ovo je osnova za matematičku aparat za računanje i izražavanje različitih matematičkih koncepta i teorema. Na klasičnoj logici temelji se i klasična (standardna) matematika. Postoje i brojni drugi logički formalni sustavi.[4] Svaki od logičkih sustava ima svoje specifične primjene i koristi se za rješavanje različitih vrsta problema. Klasična logika i matematika često su osnova za razvoj i primjenu tih drugih logičkih sustava, ali svaki od njih proširuje klasičnu logiku kako bi se bolje nosio s određenim vrstama problema ili konceptima.

3.1. Logički veznici i tablice istinitosti

Formalni jezik propozicijske logike se sastoji od:

1. Propozicijske varijable {*a*, *b*, *c*, . . . , *x*, *y*, *z*, *a*1, *a*2, *a*3, . . . , *x*1, *x*2, *x*3, . . . } - elementi skupa simbola
2. { $\neg, \vee, \wedge, \implies, \Leftrightarrow$ } - skup logičkih veznika
3. {(,)} - lijeva i desna zagrada.[4]

Svaki sud ima najmanje jedan atom koji se može vezati s drugim atomom pomoću veznika. Zagrade služe za prepoznavanje prioriteta suda. U ovom poglavlju proći ćemo kroz logičke veznike i prikazati tablice istinitosti za njih.

3.1.1. Negacija

Formula $\neg A$ je lažna ako i samo ako je formula A istinita.[4].

A	$\neg A$
$\top$	$\bot$
$\bot$	$\top$

Tablica 1: Negacija

Kod negacije se stvara suprotna vrijednost atoma.

3.1.2. Konjukcija

Formula $A \wedge B$ je istinita ako i samo ako su formula A i formula B istinite.[4]

A	B	$A \wedge B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\bot$

Tablica 2: Konjukcija

Kod konjukcije vrijedi isto kao za množenje, sud je istinit samo ako su oba atoma istinita.

3.1.3. Disjunkcija

Formula $A \vee B$ je lažna ako i samo ako su formula A i formula B neistinite.[4]

Kod disjunkcije sud vrijedi isto kao zbrajanje, sud je lažan samo ako su oba atoma lažna.

A	B	$A \vee B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

Tablica 3: Disjunkcija

3.1.4. Implikacija

Formula $A \implies B$ je lažna ako i samo ako je formula A istinita i formula B lažna.[4]

A	B	$A \implies B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\top$
$\bot$	$\bot$	$\bot$

Tablica 4: Implikacija

3.1.5. Ekvivalencija

Formula $A \Leftrightarrow B$ je istinit ako i samo ako su formula A i formula B jednake istinitosti.[4]

A	B	$A \Leftrightarrow B$
$\top$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\bot$	$\top$	$\bot$
$\bot$	$\bot$	$\top$

Tablica 5: Ekvivalencija

4. Kalkulator

U ovom poglavlju ću proći kroz izradu praktičnog dijela rada, a to je izrada kalkulatora u programskom jeziku Python i primjena kalkulatora.

4.1. Općenito o Pythonu

Python je popularan visoko pridržani programski jezik koji je razvijen krajem 1980-ih i predstavljen javnosti 1991. godine. Kreirao ga je Guido van Rossum, a od tada je postao jedan od najkorištenijih jezika u svijetu programiranja.

Programiranje (uključujući, recimo, Python programski jezik) je dio matematike, pa nas ne treba čuditi da je jezik koji bi trebao obrađivati "uobičajenu" matematiku, također može logički izraziti sve što želite o ponašanju računalnog programa.[2] Programiranje i matematika su dvije različite discipline, ali postoji snažna veza između njih. Programiranje je način primjene matematike u praksi kako bismo rješavali konkretne probleme i stvarali računalne programe. Poznat je po svojoj jednostavnosti, lakim čitanjem i pisanjem koda. U ovom radu sam koristio 2 biblioteke, a to su:

- PyParsing - za sintaktičku analizu suda
- SymPy - za operacije nad sudovima i minimizaciju suda

PyParsing je Python biblioteka koja omogućava jednostavno analiziranje teksta na temelju definiranih gramatika i pravila. Ova biblioteka omogućava programerima da lako izrade parser za obradu tekstualnih podataka, kao što su konfiguracijske datoteke, log datoteke, programski jezici ili bilo koji drugi formatirani ulaz.

Biblioteka omogućava precizno definiranje prioriteta i asociranosti operatora kako bi se osiguralo ispravno tumačenje logičkih izraza. Olakšava izradu analize različitih vrsta tekstualnih ulaza. Kroz precizno definirane gramatike i pravila, omogućava programerima da konvertiraju strukturirane tekstualne podatke u korisne strukture podataka koje se lako mogu obraditi.

Logički modul za **SymPy** omogućuje formiranje i manipuliranje logičkim izrazima pomoću simboličkih i Booleovih vrijednosti.[5]

U okviru "SymPy" biblioteke, logički sudovi i simbolička logika igraju važnu ulogu u analizi i manipulaciji matematičkih i logičkih izraza. Ovi koncepti omogućavaju programerima da rade s izjavama, tvrdnjama i logičkim operacijama na simbolički način.

SymPy je bila korisna biblioteka za rad nad sudovima. Možemo koristiti za puno više stvari kao npr. za pretvorbu u Disjunkitvnu Normalnu Formu (DNF) i Konjuktivnu Normalnu Formu (KNF), ali htio sam odraditi taj dio sam da se vidi logika iza toga. SymPy ima funkcionalnosti gdje možemo izvršavati operacije nad sudom jednostavnim linijama koda `Not`, `And`, `Or`, `Implies`, `Equivalent`.

Kod minimizacije sam koristio jednostavnu liniju koda `simplify_logic` i dobio sam minimizirani sud što mi je uvelike olakšalo posao.

4.2. Izrada kalkulatora

4.2.1. Sintaktička analiza

Programski kod za izradu sintaktičke analize logike sudova:

```
#Definiranje operatora i operanada za izvedbu sintaktičke analize logičkih izraza

akcija = ppc.integer
atom = Word(alphas, exact=1)
operand = akcija | atom

ne = oneOf("~")
i = oneOf("&")
ili = oneOf("|")
imp = oneOf(">>")
ekv = oneOf("<->")

#Definiranje gramatike za obradu sintaktičke analize logički izraza

expr = infixNotation(
    operand,
    [
        (ne, 1, opAssoc.RIGHT),
        (i, 2, opAssoc.LEFT),
        (ili, 2, opAssoc.LEFT),
        (imp, 2, opAssoc.LEFT),
        (ekv, 2, opAssoc.LEFT),
    ],
)
```

Isječak koda 1: Sintaktička analiza

U prve 3 linije koda definiramo sintaktičku analizu osnovnih operanda u izrazima. Varijabla `akcija` omogućuje analizu cjelobrojnih vrijednosti, varijabla `atom` definira atom kao jedno slovo, odnosno slova *A*,*B*,*C*... (zavisno od količine atoma u sudu), i na kraju varijabla operand daje mogućnost da vrijednost bude ili atom ili broj.

Sljedeći 5 linija definira osnovne logičke funkcije a to su redom: negacija ($\neg$), konjukcija ($\wedge$), disjunkcija ($\vee$), implikacija ($\Longrightarrow$) i ekvivalencija ($\Leftrightarrow$).

I dolazimo do najvažnijeg dijela, korištenje funkcije `infixNotation` da definira gramatiku za izradu sintaktičke analize. Glavni operandi su definirani prethodnim koracima (`operand`). Unutar funkcije pišemo asocijativnost: ne ($\neg$) se asocira s desna na lijevo (RIGHT) i ima prioritet 1; i ($\wedge$), ili ($\vee$), imp ($\Longrightarrow$) i ekv ($\Leftrightarrow$) se asociraju s lijeva na desno (LEFT) i imaju prioritet 2.

4.2.2. Izrada semantičke tablice

Prije izrade semantičke tablice slažemo sve moguće kombinacije nula i jedinica preko funkcije `kombinacije` koja prima parametar `atoms` koji sadrži broj atoma u sudu.

```
def kombinacije(atoms):
    """
    Generira sve moguće kombinacije istinitosnih vrijednosti za zadani broj atomskih izraza.

    :parametar atoms: Broj atomskih izraza za koje se generiraju kombinacije
    :tip atoms: int
    :return: Rječnik koji mapira indekse na kombinacije istinitosnih vrijednosti
    :return tip: dict
    """

    letters = [chr(ord('A') + i) for i in range(atoms)]
    combinations = list(itertools.product([False, True], repeat=atoms))
    combinations_dict = {}

    for i, kombinacija in enumerate(combinations):
        combinations_dict[i] = {letter: value for letter,
        value in zip(letters, kombinacija)}

    return combinations_dict
```

Isječak koda 2: Kombinacije

Za ovaj dio koda sam uvezao biblioteku `itertools` za kombinacije. U prvoj liniji koda stvaramo listu `letters` koja će sadržavati slova `'A', 'B', 'C', ...`ovisno o broju varijabli `atoms`. Broj elemenata u ovoj listi je jednak broju atomskih izraza, i svako slovo se generira na osnovu indeksa. Zatim kreiramo petlju koja prolazi kroz sve moguće kombinacije,

`itertools.product([False, True], repeat=atoms)` generira sve moguće kombinacije `True/False` vrijednosti za svaki atomski izraz. `repeat=atoms` prikazuje da se svaki atomski izraz ponavlja `atoms` puta, `enumerate` se koristi kako bi se dobio indeks (`i`) i trenutna kombinacija istinitosnih vrijednosti (`kombinacija`) tijekom iteracije.

Unutar petlje se stvara rječnik koji mapira svako slovo iz liste `letters` na odgovarajuću vrijednost (`True` ili `False`) iz trenutne kombinacije. Ovaj rječnik se zatim pohranjuje u rječnik `combinations_dict` s ključem `i`, gdje je `i` indeks trenutne kombinacije.

Kada dobijemo sve kombinacije prelazimo na izradu semantičke tablice. Pravimo funkciju `semanticka` koja prima sintaktičko analizirani sud i kombinacije.

```
def semanticka(sud,kombinacija):
    """
    Rekurzivno prolazi strukturu liste/sintaktičke analize `sud`
    i primjenjuje funkciju `semanticka`.

    :parametar sud: Sud u obliku sintaktičke analize
    :tip sud: list ili ParseResults
    :parametar kombinacija: Kombinacije True/False atoma i sudova.
    :tip kombinacija: dict
    """

    if sud[0]==[]:
        del sud[0]
    if(isinstance(sud[0],ParseResults)):
        semanticka(sud[0],kombinacija)
        del sud[0]
    if(len(sud)>1 and isinstance(sud[1],ParseResults)):
        semanticka(sud[1],kombinacija)
        del sud[1]
    if(len(sud)>2 and isinstance(sud[2],ParseResults)):
        semanticka(sud[2],kombinacija)
        del sud[2]
```

Isječak koda 3: Provjera instance

Na početku funkcije, naš cilj je pronaći prvu kombinaciju koju trebamo izračunati. U slučaju kada je prvi atribut u nizu analizirani rezultat umjesto atoma ili operanda, koraci se nastavljaju. U tom scenariju, iznova pozivamo istu funkciju, ali ovaj put koristeći prvi atribut u nizu kao sud. Kroz ovaj proces, dobivamo rezultat svih kombinacija za taj sud, koji će biti pohranjen u obliku ključa. Taj ključ će predstavljati kombinaciju koju smo upravo izračunali, kao na primjer $A \vee B$. Odgovarajuća vrijednost u mapiranju bit će `True` ili `False`, ovisno o rezultatu te kombinacije.

No, postoje slučajevi kada unutar analize nije prisutan prioritet u prvoj kombinaciji, kao što je situacija kada se susretnemo s izrazom poput $\neg(A \vee B)$. U takvim situacijama, sintaktička analiza stvara strukturu poput $[\neg, [A, \vee, B]]$. U ovom slučaju, drugi element tog niza, odnosno `sud[1]`, ima prioritet. Kako bismo obradili ovaj slučaj, prvo se fokusiramo na obradu `sud[1]` i izračunavanje rezultata za kombinaciju $A \vee B$. Isto tako, ista logika vrijedi i ako `sud[2]` ima prednost.

```
match sud[0]:
        """
        Prima listu `sud` i primjenjuje odgovarajuće logičke operacije na kombinacije
        iz `kombinacija`. Varijabla z predstavlja zadnji obrađeni sud,
        varijabla pz predstavlja predzadnji obrađeni sud.

        :parametar sud: Sud u obliku liste
        :tip sud: list
        :parametar kombinacija: Kombinacije True/False atoma i sudova.
        :tip kombinacija: dict
        """
        case '~':
            # Obrada negacije
            z = list(kombinacija)[-1]
            if(provjeraJedan(sud)):
                x=Not(kombinacija[sud[1]])
                kombinacija["\u00ac"+sud[1]] = x
            else:
                x=Not(kombinacija[z])
                kombinacija["\u00ac("+z+")"] = x

        case '|':
            # Obrada logičke disjunkcije
            z = list(kombinacija)[-1]
            if(provjeraJedan(sud)):
                x=Or(kombinacija[z],kombinacija[sud[1]])
                kombinacija["("+z+")\u2228"+sud[1]]=x
            else:
                pz = list(kombinacija)[-2]
                x=Or(kombinacija[pz],kombinacija[z])
                kombinacija["("+pz+")\u2228("+z+")"]=x

        case '&':
            # Obrada logičke konjunkcije
            z = list(kombinacija)[-1]
            if(provjeraJedan(sud)):
                x=And(kombinacija[z],kombinacija[sud[1]])
                kombinacija["("+z+")\u2227"+sud[1]]=x
            else:
                pz = list(kombinacija)[-2]
                x=And(kombinacija[pz],kombinacija[z])
                kombinacija["("+pz+")\u2227("+z+")"]=x

        case '>>':
            # Obrada implikacije
            z = list(kombinacija)[-1]
            if(provjeraJedan(sud)):
                x=Implies(kombinacija[z],kombinacija[sud[1]])
                kombinacija["("+z+")\u2192"+sud[1]]=x
            else:
                pz = list(kombinacija)[-2]
                x=Implies(kombinacija[pz],kombinacija[z])
                kombinacija["("+pz+")\u2192("+z+")"]=x
```


Isječak koda 3: Izrada semantičke tablice 1. dio

```
        case '<->':
            # Obrada ekvivalencije
            z = list(kombinacija)[-1]
            if(provjeraJedan(sud)):
                x=Equivalent(kombinacija[z],kombinacija[sud[1]])
                kombinacija[str(kombinacija[z])+"\u21d4"+sud[1]]=x
            else:
                pz = list(kombinacija)[-2]
                x=Equivalent(kombinacija[pz],kombinacija[z])
                kombinacija[str(kombinacija[pz])+"\u21d4"+str(kombinacija[z])]=x
```

Isječak koda 3: Izrada semantičke tablice 1. dio (nastavak)

U programskom kodu iznad provjeravamo je li na prvom mjestu operand i ako je izvršavamo izračun. U atribut `z` stavljamo zadnji dobiveni rezultat iz rječnika i to će nam trebati u slučaju da ne radimo operaciju na atom već na dobiveni rezultat iz prethodnog poziva na funkciju npr. za $[\neg, [A, B]]$ ćemo dobiti $[\neg, kombinacija]$. Funkciju koju pozivamo `provjeraJedan(sud)` provjerava je li `sud[1]` atom odnosno slovo ili je kombinacija, ako je slovo vraća `True`. Na kraju unutar kombinacije dodajemo ključ $\neg$ kombinacija (ili atom) i dajemo vrijednost `True` ili `False`.

```
def provjeraJedan(sud):
    """
    Provjerava je li drugi element suda `sud` slovo (alfanumerički karakter).
    :parametar sud: Varijabla koja se provjerava
    :tip sud: list ili str
    :return: True ako je drugi element slovo, inače False
    :return tip: bool
    """
    if(len(sud)>1 and sud[1].isalpha()):
        return True
    return False
```

Isječak koda 4: Funkcija za provjeru suda

Kod disjunkcije, konjukcije, ekvivalencije i implikacije postoji mogućnost da s obje strane operanda imamo kombinaciju umjesto atoma, pa moramo dohvaćati zadnje dvije kombinacije i na njima izvršiti operand. To možemo vidjeti kod `pz = list(kombinacija[-2])`.

```
if(len(sud)>=2):
        """
        Prima listu `sud` i primjenjuje odgovarajuće logičke operacije na kombinacije
        iz `kombinacija`. Varijabla z predstavlja zadnji obrađeni sud,
        varijabla pz predstavlja predzadnji obrađeni sud.
        Ovo ide u slučaju da se drugi element liste treba obraditi.

        :parametar sud: Sud u obliku liste
        :tip sud: list
        :parametar kombinacija: Kombinacije True/False atoma i sudova.
        :tip kombinacija: dict
        """
        match sud[1]:
            case '|':
                # Obrada logičke disjunkcije
                z = list(kombinacija)[-1]
                if(provjeraDva(sud)):
                    x=Or(kombinacija[sud[0]],kombinacija[sud[2]])
                    kombinacija[sud[0]+"\u2228"+sud[2]]=x
                else:
                    x=Or(kombinacija[sud[0]],kombinacija[z])
                    kombinacija[sud[0]+"\u2228("+z+")"]=x

            case '&':
                # Obrada logičke konjukcije
                z = list(kombinacija)[-1]
                if(provjeraDva(sud)):
                    x=And(kombinacija[sud[0]],kombinacija[sud[2]])
                    kombinacija[sud[0]+"\u2227"+sud[2]]=x
                else:
                    x=And(kombinacija[sud[0]],kombinacija[z])
                    kombinacija[sud[0]+"\u2227("+z+")"]=x

            case '>>':
                # Obrada implikacije
                z = list(kombinacija)[-1]
                if(provjeraDva(sud)):
                    x=Implies(kombinacija[sud[0]],kombinacija[sud[2]])
                    kombinacija[sud[0]+"\u2192"+sud[2]]=x
                else:
                    x=Implies(kombinacija[sud[0]],kombinacija[z])
                    kombinacija[sud[0]+"\u2192("+z+")"]=x

            case '<->':
                # Obrada ekvivalencije
                z = list(kombinacija)[-1]
                if(provjeraDva(sud)):
                    x=Equivalent(kombinacija[sud[0]],kombinacija[sud[2]])
                    kombinacija[sud[0]+"\u21d4"+sud[2]]=x
                else:
                    x=Equivalent(kombinacija[sud[0]],kombinacija[z])
                    kombinacija[sud[0]+"\u21d4("+z+")"]=x
```

Isječak koda 5: Izrada semantičke tablice 2. dio

U slučaju da je prvi atribut u nizu atom, onda se operand nalazi na drugom mjestu. Ovdje ne trebamo provjeravati negaciju jer se operand nalazi između, a ne ispred. Nakon provjere operanda radimo provjeru je li s desne strane operanda atom ili je kombinacija isto kao i u prošlom kodu, a to radimo s funkcijom `provjeraDva(sud)`. Funkcija `provjeraDva(sud)` radi isto kao i `provjeraJedan(sud)` samo što se provjerava `sud[2]`. Ako je atom onda izvršavamo operaciju sa `sud[0]` i `sud[1]`, a ako je kombinacija onda izvršavamo operaciju sa `sud[0]` i zadnjom kombinacijom (`z = list(kombinacija)[-1]`).

4.2.3. DNF i KNF

Za generiranje DNF i KNF, koristio sam gotovo rješenje koje se temelji na semantičkoj tablici. Oba ova oblika služe za izražavanje logičkih izraza na način koji olakšava analizu i manipulaciju.

Za izradu DNF-a, ključno je dohvatiti sve slučajeve u kojima je izraz istinit. To znači da trebamo identificirati sve kombinacije varijabli koje rezultiraju istinitim izrazom. Te kombinacije zatim grupiramo koristeći disjunkciju ($\vee$) kako bismo dobili DNF oblik.

Slično tome, za izradu KNF-a, potrebno je dohvatiti sve slučajeve u kojima je izraz lažan. To podrazumijeva prepoznavanje kombinacija varijabli koje dovode do lažnog izraza. Te kombinacije grupiramo pomoću konjunkcije ($\wedge$) kako bismo dobili KNF oblik.

Ovo se može jasno uočiti u donjem kodu. Kroz ovaj pristup, osiguravamo da su DNF i KNF oblici točno izraženi, uzimajući u obzir sve relevantne kombinacije varijabli.

```
for i in sud:
        """
        Iterira kroz listu i provjerava disjunktivnu i konjuktivnu normalnu formu
        za svaki element.

        :parametar x: Lista za koje se generiraju DNF i KNF oblici
        :tip x: list
        """
        if(i[list(i)[-1]]):
        #...
        else:
        #...
```

Isječak koda 6: Provjera istinitosti suda

U slučaju da je sud istinit kreiramo DNF.

```
if(i[list(i)[-1]]):
    """
    Generira Disjunktivnu Normalnu Formu (DNF) iz dane liste `sud`
    koji predstavlja logički izraz.

    :parametar sud: Lista koja predstavlja logički izraz
    :tip sud: dict
    :return: DNF oblik logičkog izraza
    :return tip: str
    """
    rez="("
    for j in range(atomi):
        if(provjera(i[list(i)[j]])):
            rez+=str(list(i)[j])
            rez+='\u2227'
        else:
            rez+='\u00ac'
            rez+=str(list(i)[j])
            rez+='\u2227'
    rez=rez.rstrip(rez[-1])
    dnff+=rez+")\u2228"
```

Isječak koda 7: Disjunktivna normalna forma

U funkciji `provjera(sud)` provjeravamo je li atom lažan ili istinit, u slučaju da je lažan dodajemo negaciju i između njih stavljamo $\wedge$ i uklanjamo zadnji $\wedge$ jer je višak. Nakon što izvršimo funkciju dodajemo $\vee$ između sudova kako bi spojili.

Isti slučaj je kod KNF samo što umjesto konjukcije ide disjunkcija u funkciji. Kod:

```
else:
    """
    Generira Konjunktivnu Normalnu Formu (KNF) iz dane liste `sud`
    koji predstavlja logički izraz.

    :parametar sud: Lista koja predstavlja logički izraz
    :tip sud: dict
    :return: KNF oblik logičkog izraza
    :return tip: str
    """
    rezz="("
    for j in range(atomi):
        if(provjera(i[list(i)[j]])):
            rezz+=str(list(i)[j])
            rezz+='\u2228'
        else:
            rezz+='\u00ac'
            rezz+=str(list(i)[j])
            rezz+='\u2228'
    rezz=rezz.rstrip(rezz[-1])
    knff+=rezz+")\u2227"
```

Isječak koda 8: Konjuktivna normalna forma

4.2.4. Logička ekvivalencija

Logička ekvivalencija se koristi za uspoređivanje i analizu dva logička izraza kako bismo utvrdili daju li iste rezultate za sve moguće kombinacije. Za provjeru logičke ekvivalencije suda korisnik upisuje drugi sud koji treba provjeriti. Nakon unošenja drugog suda provjerava se jesu li isti rezultati kod prvog i kod drugog suda. Provjeru logičke ekvivalencija izvršavamo preko for petlje koja provjerava rezultat prvog suda i uspoređuje ga s drugim sudom. Ako su isti rezultati ispisuje se da su sudovi logički jednaki. Kod:

```
def logicka_ekvivalencija(sud,atomi,sudd,atomii):
"""
Utvrđuje jesu li dvije logičke izjave ekvivalentne za zadane kombinacije atoma.

Ova funkcija uzima dvije logičke izjave 'sud' i 'sudd',
zajedno s odgovarajućim skupovima atoma 'atomi' i 'atomii'.
Izračunava izjave za sve kombinacije atoma
u odgovarajućim skupovima i provjerava jesu li rezultati ekvivalentni.

:parametar sud: Prva logička izjava
:tip sud: str
:parametar atom: Broj atoma u prvoj logičkoj izjavi
:tip atom: int
:parametar sud: Druga logička izjava
:tip sud: str
:parametar atom: Broj atoma u drugoj logičkoj izjavi
:tip atom: int
:return: Vraća True ako su sudovi ekvivalentni; False ako nisu
:return tip: bool
"""

x=[]
x2=[]
for i,kombinacija in kombinacije(atomi).items():
    x.append(semanticka(expr.parseString(sud)[0],kombinacija))
for i,kombinacija in kombinacije(atomii).items():
    x2.append(semanticka(expr.parseString(sudd)[0],kombinacija))
for i,j in zip(x,x2):
    if(i[list(i)[-1]]!=j[list(j)[-1]]):
        return False
return True
```

Isječak koda 9: Logička ekvivalencija

4.2.5. Minimizacija

Jedna od velikih prednosti korištenja modernih biblioteka za logičku analizu, poput one koju sam koristio, je efikasnost i olakšan pristup složenim postupcima. Unutar SymPy biblioteke imamo funkciju za minimizaciju suda koja koristi Quine_McCluskey algoritam. Funkcija koristi različite metode heuristike za pojednostavljanje logičkih izraza. Na kolegiju Matematika 1 obrađivali smo Veitchovu metodu minimizacije. Osnovna ideja Veitchove metode je da se formulu koja sadrži samo osnovne logičke operacije prikaže dijagramom na kojem se može uočiti jednostavnija logički ekvivalentna formula.[4]. Veitchovu metodu kasnije dorađuje Maurice Karnaugh što danas znamo kao Karnaugh mapa (K-mapa). K-mapa tako postaje korisna za funkcije sa više varijabli i postaje lakša i brža za rukovati. Quine_McCluskey algoritam je efikasniji za funkcije sa većim brojem varijabli dok Veitch dijagrami i K-mape zahtjevaju ručno grupiranje ćelija što može biti vremenski zahtjevo za funkcije s više varijabli i često se koriste kao alat za obrazovanje.

4.3. Testiranje kalkulatora

Za testiranje kalkulatora koristio sam Python testing okvir "Pytest". Moram testirati 3 funkcije a to su: `semanticka()` - provjera izrade semantičke tablice; `dnfiknff()` - provjera izrade DNF i KNF oblika; `logicka_ekvivalencija()` - provjera logičke ekvivalencije dva suda.

Za funkciju `semanticka()` trebaju 2 parametra a to su `sud` i `kombinacija`, a za rezultat ćemo dobiti semantičku tablicu u obliku rječnika u nizu.

```
@pytest.mark.parametrize("sud, kombinacija, ocekivani_rezultat", [
    (['A', '|', 'B'], {'A': False, 'B': False},

    {'A': False, 'B': False, 'AB': False}),

    (['A', '|', 'B'],

    {'A': True, 'B': True},{'A': True, 'B': True,'AB': True}),
])
def test_semanticka(sud,kombinacija,ocekivani_rezultat):
    rez = semanticka(sud,kombinacija)
    assert rez == ocekivani_rezultat
```

Isječak koda 10: Testiranje `semanticka`

Kod funkcije `dnfiknff()` kao parametre šaljemo `sud` u obliku semantičke tablice i broj atoma, a za rezultat ćemo dobiti DNF i KNF suda.

```
@pytest.mark.parametrize("sud, atomi, ocekivani_dnff, ocekivani_knff", [
    ([{'A': False, 'B': False, 'AB': False}, {'A': False, 'B': True, 'AB': True},

    {'A': True, 'B': False, 'AB': True}, {'A': True, 'B': True, 'AB': True}], 2,

    "(¬A&B)(A&¬B)(A&B)", "(¬A|¬B)"),
])
def test_dnfiknff(sud, atomi, ocekivani_dnff, ocekivani_knff):
    dnff, knff = dnfiknff(sud, atomi)
    assert knff == ocekivani_knff
```

Isječak koda 11: Testiranje `dnfiknf`

I za kraj funkcija `logicka_ekvivalencija()` prima 2 suda i broj atoma tih sudova, a za rezultat vraća bool je li sud logički ekvivalentan ili ne.

```python
@pytest.mark.parametrize("sud, atomi, sudd, atomii, ocekivani_rezultat", [
    ("A | B", 2, "B | A", 2, True),
    ("A & B", 2, "A | B", 2, False),
])
def test_logicka_ekvivalencija(sud, atomi, sudd, atomii, ocekivani_rezultat):
    rezultat = logicka_ekvivalencija(sud, atomi, sudd, atomii)
    assert rezultat == ocekivani_rezultat
```

Isječak koda 12: Testiranje `logicka_ekvivalencija`

4.4. Primjena kalkulatora

Kalkultor se pokreće preko plkal.py datoteke. Za pokrećanje python programa u naredbeni redak treba upisati naredbu: `python plkal.py "sud" -opcija`. Za ovaj dio sam koristio Python "Click" biblioteku. Za izbor imamo 8 opcija a to su: -s (–sem) služi za prikaz semantičke tablice suda; -d (–dnf) služi za prikaz DNF-a; -k (–knf) za prikaz KNF-a; –el-kon prikazuje elementarnu konjukciju suda; –el-dis prikazuje elementarnu disjunkciju suda; -t (–taut) provjerava je li sud u tautologiji; -l (–logekv) koristi se za logičku ekvivalenciju dvaju sudova koji su odvojeni znakom ekvivalencija (⇔); -m (–mini) minimizacija suda; –sve obrađuje sve opcije. Upisom naredbe `python plkal.py -help` prikazuje se sve opcije koje imaju i nazivi tih opcija.

```python
@click.command()
@click.argument('izraz', required=True)
@click.option('-s', '--sem-tablica', is_flag=True, help='Semantička tablica')
@click.option('--dnf', is_flag=True, help='DNF')
@click.option('--knf', is_flag=True, help='KNF')
@click.option('--el-kon', is_flag=True, help='Elementarna konjukcija')
@click.option('--el-dis', is_flag=True, help='Elementarna disjunkcija')
@click.option('-t','--taut', is_flag=True, help='Tautologija')
@click.option('--mini', is_flag=True, help='Minimizacija')
@click.option('--sve', is_flag=True, help='Sve')
```

Isječak koda 13: Click opcije

Kod ispisa semantičke tablice željeni sud dodajemo u listu te stvaramo prikazujemo tablicu pomoću funkcije `tabulate()` koja kao parametre prima podatke tablice i u zaglavlje stavlja ključeve to jest korak po korak kako se pravila semantička tablica.

```python
def tablica(s):
    """
    U ovom slučaju program čita string koji korisnik unosi i obrađuje
    funkciju izrada za taj sud.

    :parametar s: sud kojeg obrađujemo
    :tip s: string
    Rezultat:
    Tablica u formatu mreže koja prikazuje kombinacije ulaznog niza.
    """
    x=[]
    atomi=provjeraAtoma(s)
    for i,kombinacija in kombinacije(atomi).items():
        x.append(semanticka(expr.parseString(s)[0],kombinacija))
    kljucevi = list(x[0].keys())
    tablica = [list(y.values()) for y in x]
    print(tabulate(tablica, headers=kljucevi, tablefmt='grid'))
    pass
```

Isječak koda 14: Kod za ispis semantičke tablice

```
python plkal.py "A | B & C" -s
+-------+-------+-------+-------+-----------+
| A     | B     | C     | B&C   | A|(B&C)   |
+=======+=======+=======+=======+===========+
| False | False | False | False | False     |
+-------+-------+-------+-------+-----------+
| False | False | True  | False | False     |
+-------+-------+-------+-------+-----------+
| False | True  | False | False | False     |
+-------+-------+-------+-------+-----------+
| False | True  | True  | True  | True      |
+-------+-------+-------+-------+-----------+
| True  | False | False | False | True      |
+-------+-------+-------+-------+-----------+
| True  | False | True  | False | True      |
+-------+-------+-------+-------+-----------+
| True  | True  | False | False | True      |
+-------+-------+-------+-------+-----------+
| True  | True  | True  | True  | True      |
+-------+-------+-------+-------+-----------+
```

Isječak koda 15: Semantička tablica

Kod prikaza DNF-a i KNF-a ponovo moramo dobiti sud u obliku rječnika u nizu te zatim pozivamo funkciju dnfiknf koja vraća DNF i KNF željenog suda.

```
def dnf(s):
    """
    U ovom slučaju program čita string koji korisnik unosi i obrađuje
    funkciju dnfiknf za taj sud.

    :parametar s: sud kojeg obrađujemo
    :tip s: string
    Rezultat:
    DNF suda
    """
    x=[]
    atomi=provjeraAtoma(s)
    for i,kombinacija in kombinacije(atomi).items():
        x.append(semanticka(s,kombinacija))
    dnf,knf=dnfiknff(x,atomi)
    print("DNF: "+dnf)
    pass
```

Isječak koda 16: Kod za ispis DNF-a

```
python plkal.py "A | B & C" --dnf --knf
DNF: (¬A&B&C)|(A&¬B&¬C)|(A&¬B&C)|(A&B&¬C)|(A&B&C)
KNF: (¬A|¬B|¬C)&(¬A|¬B|C)&(¬A|B|¬C)
```

Isječak koda 17: DNF i KNF

Opcija tautologija provjerava je li sud u svakom slučaju istinit. Pozivamo funkciju `semanticka()` i provjeravamo rješenje.

```
def tau(s):
    x=[]
    atomi=provjeraAtoma(s)
    for i,kombinacija in kombinacije(atomi).items():
        x.append(semanticka(expr.parseString(s)[0],kombinacija))
    for i in x:
        if(i[list(i)[-1]]==False):
            print("Sud nije tautologija")
            return
    print("Sud je tautologija")
```

Isječak koda 18: Kod za ispis tautologije

Za minimizaciju suda pozivamo funkciju `simplify_logic()` iz Sympy biblioteke.

Korisnik ima i mogućnost da pozove više funkcija odjednom npr. semantička tablica, DNF i KNF, a to izgleda ovako:

```
python plkal.py "A | B" -s --dnf --knf
+-------+-------+-------+
| A     | B     | A|B   |
+=======+=======+=======+
| False | False | False |
+-------+-------+-------+
| False | True  | True  |
+-------+-------+-------+
| True  | False | True  |
+-------+-------+-------+
| True  | True  | True  |
+-------+-------+-------+
DNF: (¬A&B)|(A&B)
KNF: (¬A|¬B)&(A|¬B)
```

Isječak koda 19: Rezultat suda

5. Zaključak

Obrađena je teorija logike sudova, koja je osnova za razumijevanje osnovnih principa logičkog zaključivanja. Razmatrali smo različite logičke operatore, poput konjunkcije, disjunkcije, negacije i implikacije, te smo istražili njihove karakteristike i primjene. Ova teorija pruža temelj za analizu i evaluaciju različitih tvrdnji i argumentacija.

Pored toga, razvili smo kalkulator za logiku sudova koji nam omogućuje praktično primjenjivanje naučene teorije. Kroz kalkulator smo mogli unositi logičke formule, provjeravati njihovu istinitost. Ovo praktično iskustvo omogućilo nam je bolje razumijevanje kako se logika sudova primjenjuje u stvarnim situacijama.

Razumijevanje logike sudova ima široku primjenu u područjima kao što su računalna znanost, filozofija, matematika i inženjering. Kroz ovaj rad stekli smo osnovna znanja o logici sudova i razvili vještine za rješavanje logičkih problema.

U budućnosti, ovakve vještine mogu nam biti korisne za analizu i zaključivanje u različitim situacijama, te za razvijanje sofisticiranijih sistema zasnovanih na logici sudova. Kroz ovaj proces istraživanja i praktičnog rada, stvorili smo temelj koji nas potiče da nastavimo istraživati širi svijet logike, postavljajući temelje za daljnje učenje i primjenu ovog moćnog oružja razmišljanja.

Popis literature

[1] B. Divjak i T. Hunjak, *Matematika za informatičare*. Zagreb: Tiva Tiskara Varaždin, 2004., 254 str., ISBN: 953-6775-66-2.

[2] Y. A. Gonczarowski i N. Nisan, *Mathematical Logic through Python*. Cambridge: Cambridge University Press, 2022., 280 str., ISBN: 978-110-894-947-7.

[3] V. Goranko, *Logic as a Tool*. Stockholm University, Švedska: Wiley, 2016., 384 str., ISBN: 978-111-888-000-5.

[4] M. Maretić, *Uvod u matematičku logiku, skripta*. Zagreb, 2023., 44 str.

[5] „SymPy dokumentacija.” (2023.), adresa: `https://docs.sympy.org/latest/modules/logic.html` (pogledano 20. 8. 2023.).

Popis kodova

Popis tablica

1. Prilog

https://github.com/Petar47/zavrsni.git