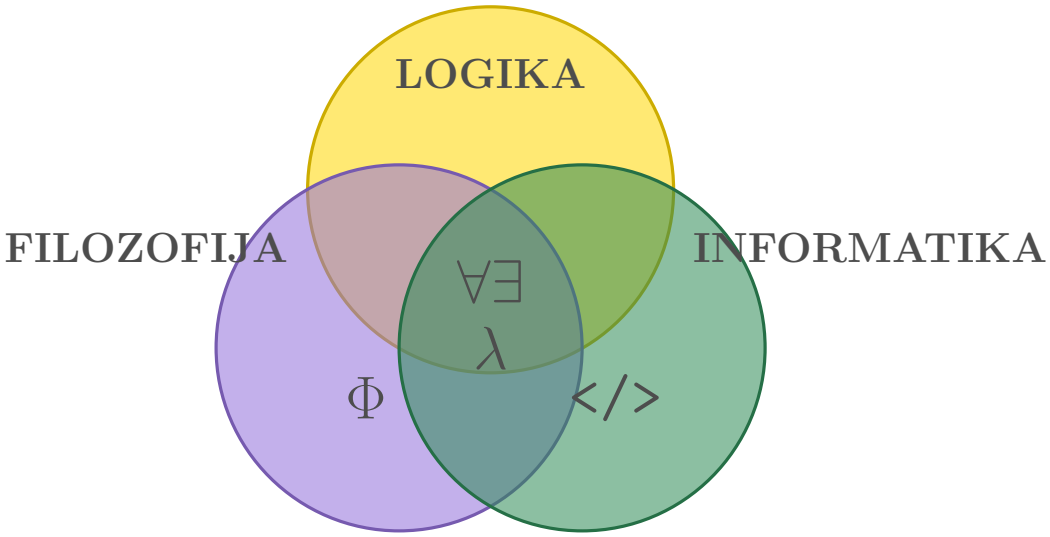
LOGIKA U KODU

Elementi logike kroz programski jezik Python



Priručnik za kolegij

„Logika i programiranje”

Davor Lauc

Filozofski fakultet

Sveučilište u Zagrebu

Nakladnik:
Sveučilište u Zagrebu
Filozofski fakultet
FF Press

Za nakladnika:
Prof. dr. sc. Domagoj Tončinić

Recenzenti:
Prof. dr. sc. Zdravko Dovedan Han
Doc. dr. sc. Ines Skelac

Grafička priprema:
Davor Lauc

ISBN:
978-953-379-265-1

DOI:
https://doi.org/10.17234/9789533792651

Prvo izdanje, Zagreb, Siječanj, 2026.

Napomena:
Svi Python primjeri koda u ovom udžbeniku dostupni su kao interaktivne Jupyter bilježnice na adresi:

`https://github.com/dlauc/logika-u-kodu`

Sadržaj

Predgovor

Ovaj udžbenik nastao je na temelju višegodišnjih predavanja na kolegiju „Logika i programiranje" na Filozofskom fakultetu Sveučilišta u Zagrebu. Kroz godine rada sa studentima filozofije koji su po prvi put pisali kod, kao i sa studentima informatike koji su otkrivali filozofske temelje svojih programa, oblikovao se pristup koji spaja apstraktno i konkretno, teoriju i praksu.

Početna ideja bila je jednostavna: učiniti formalnu logiku pristupačnijom i zanimljivijom, omogućiti transfer vještina između formalne logike i programskih vještina, te pružiti studentima filozofije dodatne „zapošljive" vještine. Umjesto da studenti samo vježbaju izvode prirodne dedukcije, zašto ih ne bi implementirali i tako bolje razumjeli? Umjesto da crtaju istinosne tablice na papiru, zašto ih ne bi generirali programski? Ono što je počelo kao istraživanje, pretvorilo se u potpuno novi način podučavanja logike.

Knjiga je organizirana u dva komplementarna dijela:

Prvi dio – Svjetovi deduktivne logike pokriva klasične sustave gdje zaključci nužno slijede iz premisa. Počinjemo s Wittgensteinovom slikom svijeta kao skupa činjenica, prelazimo na Gentzenove sustave prirodne dedukcije, istražujemo Tarskijevu semantiku, te završavamo s računalnim aspektima kroz Turingove strojeve.

Drugi dio – Svjetovi induktivnih logika istražuje sustave gdje zaključci imaju samo određeni stupanj vjerojatnosti: od Pascalove oklade preko Bayesova teorema do suvremenih pristupa u strojnom učenju.

Svako poglavlje slijedi konzistentnu strukturu: motivacija, formalizacija, implementacija, istraživanje. Ova četverodijelna struktura omogućava različite razine angažmana – od prvotnog upoznavanja do dubinskog istraživanja. Neki djelovi se djelomično preklapaju radi održavanja cjeline studenskih izlaganja.

Kao pedagoški pristup, kroz godine predavanja, razvijeno je nekoliko ključnih principa:

Greške su pedagoški momenti. Kada studentov kod ne radi, to nije neuspjeh već prilika za razumijevanje zašto logička pravila funkcioniraju kako funkcioniraju.

Apstrakcija kroz konkretno. Svaki apstraktni koncept ima konkretnu implementaciju koju je moguće pokrenuti, modificirati i igrati se s njom.

Spiralno učenje. Isti koncepti vraćaju se na različitim razinama složenosti. Implikacija se prvo pojavljuje kao Python `if-then`, zatim kao materijalna implikacija, pa kao pravilo u

prirodnoj dedukciji, i konačno kao tip funkcije, i metalogički odnos.

Na pitanje za koga je ova knjiga, odgovor je da je primarno nastala za studente filozofije koji žele razumjeti formalnu logiku kroz praktičnu primjenu. Ali kroz godine, publika se proširila:

- Studenti računarstva pronalaze filozofske temelje svoje discipline
- Nastavnici srednji škola koriste materijale za modernizaciju nastave
- Istraživači u AI-ju pronalaze korisne implementacije klasičnih sustava
- Entuzijasti koji uživaju u samostalnom istraživanju

Ne pretpostavlja se prethodno znanje programiranja – dodatak A pruža sve potrebne osnove Pythona. Također ne pretpostavlja se formalno obrazovanje iz logike – počinje se od osnova.

Kako koristiti materijale: sav kod dostupan je na `https://github.com/dlauc/logikaukodu`. Bilježnice možete:

- Pokretati lokalno s Jupyter instalacijom
- Koristiti kroz Google Colab bez instalacije
- Modificirati za vlastite potrebe
- Integrirati u vlastite kolegije

Licenca *Creative Commons* omogućava slobodno dijeljenje i adaptaciju uz navođenje izvora.

Konačno, ovaj udžbenik nije završen proizvod. Svaki semestar donosi nove uvide, nove načine objašnjavanja, nove veze između koncepata. Pozivam čitatelje da se priključe korištenjem dijeljenog koda – prijavite greške, predložite poboljšanja, podijelite svoje implementacije. Logika i programiranje nisu samo akademske discipline – oni su načini mišljenja koji oblikuju našu digitalnu stvarnost. Nadam se da će ova knjiga pomoći novim generacijama da ovladaju oboma.

Davor Lauc
Zagreb, rujan 2025

Poglavlje 1

Uvod: Što je logika i zašto kod?

> Ne priliči izvrsnim ljudima da kao robovi gube sate na računanje, u poslu koji se može s punim povjerenjem prepustiti bilo kome drugome uporabom strojeva.
>
> Gottfried Wilhelm Leibniz[1]

1.1 Logika kao znanost o valjanom zaključivanju

Logika proučava oblike valjanog zaključivanja i relacije logičkog slijeda. Kada kažemo da je zaključak valjan, mislimo da konkluzija nužno slijedi iz premisa. Ova nužnost proizlazi iz same strukture našeg mišljenja, neovisno o sadržaju.

Razmotrimo klasičan primjer:

1. Svi ljudi su smrtni.
2. Sokrat je čovjek.
3. Dakle, Sokrat je smrtan.

Valjanost ovog zaključka ne ovisi o tome tko je Sokrat ili što znači čovjek" i smrtan". Ona proizlazi iz logičke forme koja ostaje valjana čak i kad zamijenimo termine:

1. Svi P su Q.
2. a je P.
3. Dakle, a je Q.

[1] *Machina Arithmetica in qua non Aditio tantum Subtractio 1685*, slobodni prijevod s engleskog prijevoda

Moderna simbolička logika omogućava nam precizno zapisivanje ovakvih formi. U logici sudova bavimo se veznicima:

- Konjunkcija ($\wedge$): „i”
- Disjunkcija ($\vee$): „ili”
- Implikacija ($\rightarrow$): „ako...onda”
- Negacija ($\neg$): „nije”

Logika predikata ide dalje omogućavajući kvantifikaciju:

- Univerzalni kvantifikator ($\forall$): „svi”, „svaki”
- Egzistencijalni kvantifikator ($\exists$): „neki”, „postoji”

1.2 Formalni jezik misli

Gottlob Frege, utemeljitelj moderne logike, nastojao je stvoriti ..pojmovno pismo” (*Begriffsschrift*) – formalni jezik za izražavanje čistog mišljenja, oslobođenog dvosmislenosti prirodnog jezika. Njegova vizija danas živi u programskim jezicima.

Kada pišemo Python kod:

```
def je_smrtan(x):
    return je_čovjek(x)

assert je_smrtan("Sokrat") == True
```

formaliziramo logičku strukturu. Funkcija `je_smrtan` enkodira univerzalnu tvrdnju, a `assert` provjerava konkretnu instancu.

1.3 Struktura stvarnosti: Wittgensteinova slika

Ludwig Wittgenstein u *Tractatus Logico-Philosophicus* nudi radikalnu tezu: granice našeg jezika su granice našeg svijeta. Za njega, svijet je totalitet činjenica, ne stvari. Činjenice postoje u ..logičkom prostoru” – prostoru svih mogućih kombinacija.

Ova ideja ima izravnu programsku interpretaciju. Kada definiramo Booleove varijable:

```
kiša = True
sunce = False
vjetar = True
```

definiramo točku u logičkom prostoru. S tri varijable, imamo $2^3 = 8$ mogućih svjetova. Svaki program implicitno definira takav prostor mogućnosti i pravila kretanja kroz njega.

1.4 Istina i značenje: Tarskijeva semantika

Alfred Tarski riješio je drevni problem istine kroz svoju semantičku teoriju. Njegova čuvena T-shema:

..Snijeg je bijel" je istinito ako i samo ako snijeg je bijel.

Može se činiti trivijalnom, ali razlikuje jezik od metajezika. U Pythonu:

```
def je_istinito(izjava, model):
    return eval(izjava, model)

model = {"snijeg_je_bijel": True}
assert je_istinito("snijeg_je_bijel", model) == True
```

funkcija `je_istinito` je metajezična – ona govori *o* izjavama, ne *u* njima.

1.5 Dokazi kao programi: Curry-Howard izomorfizam

Jedan od najdubljih uvida 20. stoljeća je otkriće strukturne ekvivalencije između logičkih dokaza i programa. Curry-Howard izomorfizam pokazuje:

Logika	**Programiranje**
Formula $A \to B$	Tip funkcije `A -> B`
Dokaz formule	Program tog tipa
Modus ponens	Aplikacija funkcije
Konjunkcija $A \wedge B$	Tuple `(A, B)`
Disjunkcija $A \vee B$	Union type `A | B`

Svaki put kad pišete funkciju, vi zapravo konstruirate dokaz. Svaki put kad je pozivate, primjenjujete logičko pravilo. Tipovi su teoremi, programi su dokazi!

1.6 Granice formalizma: Gödelova nepotpunost

Kurt Gödel je 1931. dokazao da svaki dovoljno bogat formalni sustav ili je nekonzistentan ili nepotpun. Postoje istinite tvrdnje koje se ne mogu dokazati unutar sustava.

Njegov dokaz koristi samoreferenciranje – konstruira rečenicu koja kaže ..Ja nisam dokaziva". Ako je dokaziva, sustav je nekonzistentan. Ako nije, sustav je nepotpun.

U Pythonu možemo ilustrirati Gödelov trik:

```
def gödel_rečenica(n):
```

```
    """Rečenica koja tvrdi da nije dokaziva"""
    return f"Rečenica broj {n} nije dokaziva"


\textbf{Paradoks: ako je dokaziva, onda je lažna}

\textbf{Ako nije dokaziva, onda je istinita, ali nedokaziva}
```

1.7 Turingovi strojevi

Alan Turing definirao je precizni model računanja – Turingov stroj. Pokazao je da postoje problemi koje nijedan stroj ne može riješiti, poput problema zaustavljanja.

Python interpreter je zapravo Turingov stroj (tehnički Turing potpun jezik). Svaki program koji napišete je opis konačnog automata koji manipulira simbolima na ..traci" (memoriji):

```
def turingov_stroj(traka, program, stanje=0):
    while stanje != "STOP":
        simbol = traka.pročitaj()
        novo_stanje, novi_simbol, smjer = program[stanje][simbol]
        traka.zapiši(novi_simbol)
        traka.pomakni(smjer)
        stanje = novo_stanje
    return traka
```

1.8 Od dedukcije k indukciji

Klasična logika bavi se nužnim zaključcima. Ali većina našeg zaključivanja je probabilistička. David Hume primijetio je „problem indukcije" – iz činjenice da je sunce izašlo svaki dan do sada, ne slijedi nužno da će izaći sutra.

Bayesov teorem daje nam formalni okvir za induktivno zaključivanje:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

Gdje je $P(H|E)$ posteriorna vjerojatnost hipoteze nakon evidencije.

U Pythonu:

```
def bayes(prior, likelihood, evidence):
return (likelihood * prior) / evidence
```

```
\textbf{Medicinska dijagnoza}

p_bolest = 0.01  # 1% populacije ima bolest
p_pozitivan_test_ako_bolest = 0.99  # 99% osjetljivost
p_pozitivan_test = 0.05  # 5% ukupno pozitivnih


p_bolest_ako_pozitivan = bayes(
p_bolest,
p_pozitivan_test_ako_bolest,
p_pozitivan_test
)


\textbf{Rezultat: 0.198 ili 19.8%}
```

1.9 Paradoksi i granice

Logika je puna paradoksa koji testiraju naše razumijevanje:

Russellov paradoks: Skup svih skupova koji ne sadrže sebe. Sadrži li sebe?

Paradoks lažljivca: ..Ova rečenica je neistinita." Istinita ili neistinita?

Sorites paradoks: Jedna zrna pijeska nije hrpa. Dodavanje jednog zrna ne čini hrpu. Dakle, nikad nema hrpe?

Ovi paradoksi nisu samo zagonetke – oni otkrivaju fundamentalne limite formalnih sustava i potiču razvoj novih logika (parakontistentne, fuzzy, relevantne).

1.10 Primjena u stvarnom svijetu

Logika kroz kod nije samo akademska vježba. Primjene su svugdje:

Baze podataka koriste logiku predikata (SQL je zapravo logički jezik):

```
SELECT * FROM studenti
WHERE godina > 2 AND prosjek >= 4.0
```

Verifikacija softvera dokazuje korektnost kritičnih sustava:

```
@requires(x >= 0)
@ensures(rezultat >= 0)
```

```
def korijen(x):
    return sqrt(x)
```

Umjetna inteligencija koristi logičko zaključivanje za planiranje i donošenje odluka.

1.11 Putovanje koje slijedi

Ovaj udžbenik vodi vas kroz postupno otkrivanje veze između logike i programiranja. Svaki koncept gradit ćemo od temelja:

1. **Intuicija**: Zašto je koncept važan?
2. **Formalizacija**: Precizna matematička definicija
3. **Implementacija**: Radni Python kod
4. **Eksploracija**: Eksperimenti i varijacije

Ne učite samo *o* logici – prakticirajte logiku kroz kod. Svaka Jupyter bilježnica je laboratorij. Mijenjajte parametre, testirajte granične slučajeve, pokušajte pokvariti kod. Kroz ove eksperimente razvit ćete dublju intuiciju od pukog čitanja.

Zapamtite: u logici, kao i u programiranju, jedini način učenja je činjenjem. Greške su dragocjene – one otkrivaju skrivene pretpostavke i suptilne istine.

Započnimo ovo putovanje kroz svjetove logike, gdje svaki novi koncept otvara vrata dubljeg razumijevanja načina na koji mislimo, zaključujemo i stvaramo.

Dio I

SVJETOVI DEDUKTIVNE LOGIKE

Poglavlje 2

Wittgensteinova logička slika svijeta

2.1 Semantička logička posljedica kroz prizmu *Tractatusa*

U ovom poglavlju istražujemo koncept **semantičke logičke posljedice** kroz praktičnu Python implementaciju, inspirirani Wittgensteinovim pristupom logici i ontologiji iz *Tractatus Logico-Philosophicus*.

> "Svijet je sve što je slučaj" (*Die Welt ist alles, was der Fall ist*) - Tractatus, 1

Ova slavna prva rečenica *Tractatusa* postavlja temelje za razumijevanje odnosa između jezika, logike i stvarnosti.

2.1.1 Atomarne činjenice i elementarni sudovi

Za Wittgensteina, svijet se sastoji od **činjenica** (*Tatsachen*), ne stvari:

> "Svijet je totalitet činjenica, ne stvari" - Tractatus, 1.1

Atomarne činjenice (*Sachverhalte*) su najjednostavnije činjenice koje mogu postojati ili ne postojati. U logici ih predstavljamo **elementarnim sudovima**.

```python
class ElementarniSud:
    """Predstavlja atomarnu činjenicu u Wittgensteinovom smislu."""

    def __init__(self, simbol, opis=""):
        self.simbol = simbol
        self.opis = opis
        self.vrijednost = None
```

```

    def __repr__(self):
        return f"{self.simbol}: {self.opis}"

    def __bool__(self):
        return bool(self.vrijednost)

# Stvorimo elementarne sudove koji odgovaraju atomarnim činjenicama
p = ElementarniSud("p", "Pada kiša")
q = ElementarniSud("q", "Ulice su mokre")
r = ElementarniSud("r", "Nosim kišobran")

print("Elementarni sudovi (atomarne činjenice):")
print(f"  {p}")
print(f"  {q}")
print(f"  {r}")
```

Output

```
Elementarni sudovi (atomarne činjenice):
  p: Pada kiša
  q: Ulice su mokre
  r: Nosim kišobran
```

2.1.2 Logički prostor i mogući svjetovi

Wittgenstein uvodi pojam **logičkog prostora** kao totaliteta svih mogućih konfiguracija atomarnih činjenica:

> "Činjenice u logičkom prostoru jesu svijet" - Tractatus, 1.13

Svaki **mogući svijet** je jedna određena kombinacija postojanja i nepostojanja atomarnih činjenica.

```
import itertools

def generiraj_moguce_svjetove(elementarni_sudovi):
    """Generira sve moguće svjetove kao kombinacije istinitosnih vrijednosti."""
    svjetovi = []
    n = len(elementarni_sudovi)

    for vrijednosti in itertools.product([False, True], repeat=n):
        svijet = {}
        for sud, vrijednost in zip(elementarni_sudovi, vrijednosti):
            svijet[sud.simbol] = vrijednost
        svjetovi.append(svijet)
```

```
13
14      return svjetovi
15
16  # Generiraj logički prostor
17  sudovi = [p, q, r]
18  svi_svjetovi = generiraj_moguce_svjetove(sudovi)
19
20  print(f"Logički prostor sadrži {len(svi_svjetovi)} mogućih svjetova:\n")
21  for i, svijet in enumerate(svi_svjetovi[:4], 1):  # Prikaži prve 4
22      # Koristi simbole ⊤ i ⊥ umjesto True/False
23      svijet_prikaz = {k: '⊤' if v else '⊥' for k, v in svijet.items()}
24      print(f"Svijet {i}: {svijet_prikaz}")
25  print("...")
```

Output

```
Logički prostor sadrži 8 mogućih svjetova:

Svijet 1: {'p': '⊥', 'q': '⊥', 'r': '⊥'}
Svijet 2: {'p': '⊥', 'q': '⊥', 'r': '⊤'}
Svijet 3: {'p': '⊥', 'q': '⊤', 'r': '⊥'}
Svijet 4: {'p': '⊥', 'q': '⊤', 'r': '⊤'}
...
```

2.1.3 Logički veznici i složeni sudovi

Wittgenstein pokazuje kako se složeni sudovi grade iz elementarnih pomoću **istinosno-funkcionalnih veznika**:

> "Sud je istinosna funkcija elementarnih sudova" - Tractatus, 5

Implementirajmo osnovne logičke veznike koristeći Python operatore:

```
1   class Sud:
2       """Predstavlja sud koji može biti elementaran ili složen."""
3
4       def __init__(self, formula, opis=""):
5           self.formula = formula
6           self.opis = opis
7           self.evaluacija = None
8
9       def evaluiraj(self, svijet):
10          """Evaluira sud u danom mogućem svijetu."""
11          # Ovdje bi trebala biti logika evaluacije
12          # Za sada vraćamo jednostavnu vrijednost
13          return self.evaluacija if self.evaluacija is not None else False
```

```python

    def __repr__(self):
        return self.formula

# Definiraj funkcije za logičke veznike
def negacija(sud):
    """Negacija: ¬p"""
    return Sud(f"¬{sud.formula}", f"nije slučaj da {sud.opis}")

def konjunkcija(sud1, sud2):
    """Konjunkcija: p ∧ q"""
    return Sud(f"({sud1.formula} ∧ {sud2.formula})",
               f"{sud1.opis} i {sud2.opis}")

def disjunkcija(sud1, sud2):
    """Disjunkcija: p ∨ q"""
    return Sud(f"({sud1.formula} ∨ {sud2.formula})",
               f"{sud1.opis} ili {sud2.opis}")

def implikacija(sud1, sud2):
    """Materijalna implikacija: p → q"""
    return Sud(f"({sud1.formula} → {sud2.formula})",
               f"ako {sud1.opis}, onda {sud2.opis}")

# Primjeri složenih sudova
p_sud = Sud("p", "pada kiša")
q_sud = Sud("q", "ulice su mokre")

složeni1 = implikacija(p_sud, q_sud)
složeni2 = konjunkcija(p_sud, negacija(q_sud))

print("Složeni sudovi:")
print(f"  {složeni1}: {složeni1.opis}")
print(f"  {složeni2}: {složeni2.opis}")
```

Output

```
Složeni sudovi:
  (p → q): ako pada kiša, onda ulice su mokre
  (p ∧ ¬q): pada kiša i nije slučaj da ulice su mokre
```

2.1.4 Semantička logička posljedica

Ključni koncept u logici je **logička posljedica** (semantička implikacija). Kažemo da je sud φ logička posljedica skupa sudova Γ, što zapisujemo:

$$\Gamma \models \varphi$$

ako i samo ako u **svakom mogućem svijetu** gdje su svi sudovi iz Γ istiniti, φ je također istinit.

Za Wittgensteina, logička posljedica pokazuje strukturalni odnos između sudova:

> "Kada istinitost jednog suda slijedi iz istinitosti drugih, to vidimo iz strukture samih sudova" - Tractatus, 5.13

```python
def je_logicka_posljedica(premisa, zakljucak, elementarni):
    """
    Provjerava je li zaključak logička posljedica premise.

    Args:
        premisa: funkcija koja evaluira premisu
        zakljucak: funkcija koja evaluira zaključak
        elementarni: lista elementarnih sudova

    Returns:
        (bool, protuprimjer ili None)
    """
    for vrijednosti in itertools.product([False, True], repeat=len(elementarni)):
        # Postavi vrijednosti elementarnih sudova
        svijet = dict(zip(elementarni, vrijednosti))

        # Evaluiraj premisu i zaključak
        p_istina = premisa(svijet)
        z_istina = zakljucak(svijet)

        # Ako premisa istinita a zaključak lažan - nije logička posljedica
        if p_istina and not z_istina:
            return False, svijet

    return True, None

# Definiraj jednostavne evaluacijske funkcije
def p_eval(svijet):
    return svijet.get('p', False)

def p_imp_q(svijet):
    p = svijet.get('p', False)
    q = svijet.get('q', False)
    return not p or q  # p → q ⇔ ¬p ∨ q

def q_eval(svijet):
    return svijet.get('q', False)

# Test: Je li q logička posljedica od (p → q) ∧ p? (Modus Ponens)
def modus_ponens_premisa(svijet):
    return p_imp_q(svijet) and p_eval(svijet)

```

```
rezultat, protuprimjer = je_logicka_posljedica(
    modus_ponens_premisa,
    q_eval,
    ['p', 'q']
)

print("Test Modus Ponens: ((p → q) ∧ p) ⊨ q")
if rezultat:
    print("  ✓ JEST logička posljedica")
else:
    print(f"  × NIJE logička posljedica. Protuprimjer: {protuprimjer}")
```

Output

```
Test Modus Ponens: ((p → q) ∧ p) ⊨ q
  ✓ JEST logička posljedica
```

2.1.5 Tautologije i kontradikcije

Wittgenstein ističe poseban status **tautologija** i **kontradikcija**:

> "Tautologija i kontradikcija nisu slike stvarnosti. One ne predstavljaju nikakvu moguću situaciju" - Tractatus, 4.462

- **Tautologija**: istinita u svim mogućim svjetovima (npr. $p \vee \neg p$)
- **Kontradikcija**: lažna u svim mogućim svjetovima (npr. $p \wedge \neg p$)

One pokazuju granice logičkog prostora:

```
def provjeri_status(formula_eval, elementarni):
    """Provjerava je li formula tautologija, kontradikcija ili kontingentna."""
    istiniti = 0
    ukupno = 0

    for vrijednosti in itertools.product([False, True], repeat=len(elementarni)):
        svijet = dict(zip(elementarni, vrijednosti))
        if formula_eval(svijet):
            istiniti += 1
        ukupno += 1

    if istiniti == ukupno:
        return "TAUTOLOGIJA"
    elif istiniti == 0:
        return "KONTRADIKCIJA"
```

```
    else:
        return f"KONTINGENTNA ({istiniti}/{ukupno} svjetova)"

# Testiraj različite formule
def tautologija(svijet):
    p = svijet.get('p', False)
    return p or not p  # p ∨ ¬p

def kontradikcija(svijet):
    p = svijet.get('p', False)
    return p and not p  # p ∧ ¬p

def kontingentna(svijet):
    return svijet.get('p', False)  # samo p

print("Status formula u logičkom prostoru:\n")
print(f"p ∨ ¬p: {provjeri_status(tautologija, ['p'])}")
print(f"p ∧ ¬p: {provjeri_status(kontradikcija, ['p'])}")
print(f"p: {provjeri_status(kontingentna, ['p'])}")
```

Output

```
Status formula u logičkom prostoru:

p ∨ ¬p: TAUTOLOGIJA
p ∧ ¬p: KONTRADIKCIJA
p: KONTINGENTNA (1/2 svjetova)
```

2.1.6 Tablični prikaz logičkih posljedica

Vizualizirajmo logičke posljedice pomoću **tablica istinitosti**, što odgovara Wittgensteinovoj metodi iz *Tractatusa*:

```
def tablica_istinitosti(premise, zakljucak, varijable):
    """Generira tablicu istinitosti za provjeru logičke posljedice."""
    print("\nTablica istinitosti:")
    print("=" * 60)

    # Zaglavlje
    header = " | ".join(varijable) + " | Premisa | Zaključak | Status"
    print(header)
    print("-" * len(header))

    je_posljedica = True

    for vrijednosti in itertools.product([" ⊥ ", " ⊤ "], repeat=len(varijable)):
        svijet = {var: (val.strip() == "⊤") for var, val in zip(varijable, vrijednosti)}

```

```
        p_val = premise(svijet)
        z_val = zakljucak(svijet)

        # Provjeri je li narušena logička posljedica
        status = ""
        if p_val and not z_val:
            status = "<-- PROTUPRIMJER"
            je_posljedica = False

        # Ispis reda
        row = " | ".join(vrijednosti)
        row += f" |    {'⊤' if p_val else '⊥'}    |    {'⊤' if z_val else '⊥'}    |
        ↪ {status}"
        print(row)

    print("=" * 60)
    if je_posljedica:
        print("\n✓ Zaključak JEST logička posljedica premise")
    else:
        print("\n× Zaključak NIJE logička posljedica premise")

    return je_posljedica

# Test klasičnih logičkih zakona
print("\nMODUS PONENS: ((p → q) ∧ p) ⊨ q")
tablica_istinitosti(modus_ponens_premisa, q_eval, ['p', 'q'])

# Test disjunktivnog silogizma
def disj_silogizam_premisa(svijet):
    p = svijet.get('p', False)
    q = svijet.get('q', False)
    return (p or q) and not p

print("\nDISJUNKTIVNI SILOGIZAM: ((p ∨ q) ∧ ¬p) ⊨ q")
tablica_istinitosti(disj_silogizam_premisa, q_eval, ['p', 'q'])
```

Output

```
MODUS PONENS: ((p → q) ∧ p) ⊨ q

Tablica istinitosti:
============================================================
p | q | Premisa | Zaključak | Status
------------------------------------
 ⊥  |  ⊥  |    ⊥    |    ⊥    |
 ⊥  |  ⊤  |    ⊥    |    ⊤    |
 ⊤  |  ⊥  |    ⊥    |    ⊥    |
 ⊤  |  ⊤  |    ⊤    |    ⊤    |
============================================================

✓ Zaključak JEST logička posljedica premise
```

```
DISJUNKTIVNI SILOGIZAM: ((p ∨ q) ∧ ¬p) ⊨ q

Tablica istinitosti:
============================================================
p | q | Premisa | Zaključak | Status
------------------------------------
 ⊥  |  ⊥  |    ⊥    |     ⊥     |
 ⊥  |  ⊤  |    ⊤    |     ⊤     |
 ⊤  |  ⊥  |    ⊥    |     ⊥     |
 ⊤  |  ⊤  |    ⊥    |     ⊤     |
============================================================

✓ Zaključak JEST logička posljedica premise
True
```

2.1.7 Filozofske implikacije

Granice jezika i logike

Wittgenstein pokazuje da logika ima svoje granice:

> "Logika ispunjava svijet; granice svijeta su također njezine granice" - Tractatus, 5.61

Naša implementacija demonstrira ove granice:

- **Tautologije** ne govore ništa o svijetu (istinite su uvijek)
- **Kontradikcije** opisuju nemogućnosti
- Samo **kontingentne formule** zapravo opisuju moguća stanja svijeta

Slikovna teorija značenja

Prema Wittgensteinu, sudovi su **slike mogućih stanja stvari**:

> "Logička slika činjenica jest misao" - Tractatus, 3

Naš kod modelira ovu ideju:

- Elementarni sudovi = atomarne činjenice
- Složeni sudovi = kombinacije atomarnih činjenica
- Mogući svjetovi = sve moguće konfiguracije

2.1.8 Praktična primjena: Logičko zaključivanje

Implementirajmo jednostavan sustav za automatsko logičko zaključivanje:

```python
class LogickiSustav:
    """Jednostavan sustav za logičko zaključivanje."""

    def __init__(self):
        self.baza_znanja = []
        self.elementarni = set()

    def dodaj_premisu(self, formula, varijable):
        """Dodaje premisu u bazu znanja."""
        self.baza_znanja.append(formula)
        self.elementarni.update(varijable)

    def moze_zakljuciti(self, zakljucak):
        """Provjerava slijedi li zaključak iz baze znanja."""

        def premise_eval(svijet):
            # Sve premise moraju biti istinite
            for premisa in self.baza_znanja:
                if not premisa(svijet):
                    return False
            return True

        # Provjeri sve moguće svjetove
        for vrijednosti in itertools.product([False, True],
                                             repeat=len(self.elementarni)):
            svijet = dict(zip(list(self.elementarni), vrijednosti))

            if premise_eval(svijet) and not zakljucak(svijet):
                return False, svijet  # Našli protuprimjer

        return True, None

# Primjer korištenja
sustav = LogickiSustav()

# Dodaj premise
def ako_kisa_mokro(svijet):
    return not svijet.get('kiša', False) or svijet.get('mokro', False)

def kisa_pada(svijet):
    return svijet.get('kiša', False)

sustav.dodaj_premisu(ako_kisa_mokro, ['kiša', 'mokro'])
sustav.dodaj_premisu(kisa_pada, ['kiša'])

# Testiraj zaključke
def ulice_mokre(svijet):
```

```
    return svijet.get('mokro', False)

rezultat, protuprimjer = sustav.moze_zakljuciti(ulice_mokre)

print("Baza znanja:")
print("  1. Ako pada kiša, ulice su mokre")
print("  2. Pada kiša")
print("\nZaključak: Ulice su mokre")
print(f"\nRezultat: {'✓ Logički slijedi' if rezultat else '× Ne slijedi'}")
```

2.1.9 Zadaci i prijedlozi za daljnje istraživanje

Za produbljivanje razumijevanja semantičke logičke posljedice i Wittgensteinove filozofije, predlažemo sljedeće istraživačke teme prikladne za dodiplomske studente:

2.1.10 Prijedlozi za daljnje istraživanje

Za produbljivanje razumijevanja semantičke logičke posljedice kroz praktične Python zadatke, predlažemo sljedeće vježbe prikladne za studente koji uče osnove logike sudova:

Proširenje skupa logičkih veznika

Implementirajte funkcije za dodatne logičke veznike: ekskluzivnu disjunkciju (XOR), Shefferovu crticu (NAND) i Pierceovu strelicu (NOR). Pokažite da su NAND i NOR funkcionalno potpuni - da pomoću njih možete izraziti sve ostale veznike.

Automatska generacija tablica istinitosti

Napišite funkciju koja prima proizvoljan logički izraz kao string (npr. "(p → q) ∧ (q → r)") i automatski generira njegovu tablicu istinitosti. Koristite Python `eval()` funkciju uz sigurnosne provjere.

Prepoznavanje tautologija

Stvorite funkciju koja provjerava je li dana formula tautologija bez generiranja cijele tablice istinitosti - zaustavite se čim nađete protuprimjer. Testirajte na klasičnim zakonima: De Morganovim zakonima, zakonu distribucije, zakonu kontrapozicije.

Normalne forme

Implementirajte pretvorbu formula u konjunktivnu (KNF) i disjunktivnu (DNF) normalnu formu. Za danu tablicu istinitosti generirajte minimalnu formulu koja je opisuje.

Provjera ekvivalencije

Napišite funkciju koja provjerava jesu li dvije formule logički ekvivalentne. Testirajte s primjerima poput: je li $(p \to q)$ ekvivalentno s $(\neg p \vee q)$? Je li $(p \to (q \to r))$ ekvivalentno s $((p \wedge q) \to r)$?

Broj mogućih formula

Za n propozicijskih varijabli, koliko različitih (neekvivalentnih) formula možete stvoriti? Napišite program koji generira sve moguće tablice istinitosti za 2 i 3 varijable i broji koliko ih je jedinstvenih.

Vizualizacija logičkih odnosa

Koristeći matplotlib, nacrtajte graf gdje čvorovi predstavljaju moguće svjetove, a bridovi povezuju svjetove koji se razlikuju u točno jednoj atomarnoj činjenici. Obojite svjetove gdje je vaša formula istinita.

Minimalni skup premisa

Za dani zaključak i skup premisa, pronađite minimalni podskup premisa iz kojeg zaključak još uvijek slijedi. Na primjer, ako imate premise $p, p \to q, q \to r, p \to r$, koji je minimalni skup za zaključak r?

Interaktivni dokazivač

Stvorite jednostavnu interaktivnu aplikaciju gdje korisnik može graditi dokaz korak po korak koristeći osnovna pravila (modus ponens, modus tollens, disjunktivni silogizam). Program provjerava valjanost svakog koraka.

Analiza složenosti formula

Napišite funkcije koje mjere "složenost" formule: broj veznika, dubinu ugniježđenja, broj različitih varijabli. Istražite odnos između složenosti formule i broja redaka u njenoj minimalnoj DNF reprezentaciji.

Svaki zadatak postupno gradi razumijevanje ključnih koncepata semantičke logičke posljedice kroz praktično programiranje, omogućavajući studentima da eksperimentiraju s logičkim strukturama i razviju intuiciju za formalno zaključivanje.

2.1.11 Zaključak

Kroz ovu implementaciju istražili smo temeljne koncepte semantičke logičke posljedice inspirirani Wittgensteinovom filozofijom:

1. **Atomarne činjenice** kao građevni blokovi stvarnosti
2. **Logički prostor** kao totalitet mogućih svjetova
3. **Logička posljedica** kao odnos koji vrijedi u svim mogućim svjetovima
4. **Granice logike** pokazane kroz tautologije i kontradikcije

Wittgensteinov zaključak *Tractatusa* podsjeća nas:

> "O čemu se ne može govoriti, o tome se mora šutjeti" - Tractatus, 7

Logika može opisati strukturu mogućih svjetova, ali sama ta sposobnost počiva na metalogičkim osnovama koje se ne mogu izraziti unutar sustava. Naš Python kod demonstrira ovu granicu - možemo implementirati logiku, ali pitanje zašto logika funkcionira ostaje izvan dosega same logike.

Kroz praktično programiranje otkrivamo da je razumijevanje logičke posljedice ključno ne samo za filozofiju jezika i logike, već i za moderna područja poput verifikacije softvera, umjetne inteligencije i automatskog zaključivanja.

Poglavlje 3

Gentzenov svijet: Prirodna dedukcija i sintaktička logička posljedica

3.1 Od semantike k sintaksi

Dok je Wittgenstein u *Tractatusu* istraživao semantičke temelje logike kroz pojam mogućih svjetova, Gerhard Gentzen (1909-1945) revolucionirao je logiku uvođenjem **prirodne dedukcije** - sustava koji formalizira kako zapravo zaključujemo.

> "Moja polazna točka bila je sljedeća: logički izračun, kako se danas prezentira, odvija se, takoreći, u jednom potopljenom svijetu, svijetu logičkih formula, koji uopće nije prirodan za razumijevanje" - Gentzen, 1935

Gentzenov pristup vraća logiku njezinim korijenima - ljudskom zaključivanju.

3.1.1 Sintaktička naspram semantičke logičke posljedice

Gottlob Frege, utemeljitelj moderne logike, prvi je jasno razlikovao **sadržaj** od **forme** zaključivanja:

> "Der waagerechte Strich, aus dem das Zeichen zusammengesetzt ist, verbindet die ihm folgenden Zeichen zu einem Ganzen, und auf dieses Ganze bezieht sich die durch den senkrechten Strich am linken Ende des waagerechten ausgedrückte Bejahung. Der waagerechte Strich mag der Inhaltsstrich, der senkrechte der Urteilsstrich heissen." - Frege, *Begriffsschrift*, 1879

Ova distinkcija vodi nas k razlikovanju:

- **Semantička logička posljedica** ($\Gamma \models \varphi$): istinitost u svim modelima

- **Sintaktička logička posljedica** ($\Gamma \vdash \varphi$): izvođenje pomoću pravila

Implementirajmo oba pristupa:

```python
from dataclasses import dataclass
from typing import List, Set, Optional, Tuple
from enum import Enum

class TipFormule(Enum):
    """Tipovi logičkih formula."""
    ATOM = "atom"
    NEGACIJA = "¬"
    KONJUNKCIJA = "∧"
    DISJUNKCIJA = "∨"
    IMPLIKACIJA = "→"

@dataclass
class Formula:
    """Predstavlja logičku formulu u sintaktičkom obliku."""
    tip: TipFormule
    sadrzaj: any  # atom: string, ostalo: tuple formula

    def __repr__(self):
        if self.tip == TipFormule.ATOM:
            return self.sadrzaj
        elif self.tip == TipFormule.NEGACIJA:
            return f"¬{self.sadrzaj}"
        elif self.tip in [TipFormule.KONJUNKCIJA, TipFormule.DISJUNKCIJA,
        ↪  TipFormule.IMPLIKACIJA]:
            lijevo, desno = self.sadrzaj
            return f"({lijevo}{self.tip.value}{desno})"

    def evaluiraj(self, model):
        """Semantička evaluacija formule u modelu."""
        if self.tip == TipFormule.ATOM:
            return model.get(self.sadrzaj, False)
        elif self.tip == TipFormule.NEGACIJA:
            return not self.sadrzaj.evaluiraj(model)
        elif self.tip == TipFormule.KONJUNKCIJA:
            l, d = self.sadrzaj
            return l.evaluiraj(model) and d.evaluiraj(model)
        elif self.tip == TipFormule.DISJUNKCIJA:
            l, d = self.sadrzaj
            return l.evaluiraj(model) or d.evaluiraj(model)
        elif self.tip == TipFormule.IMPLIKACIJA:
            l, d = self.sadrzaj
            return not l.evaluiraj(model) or d.evaluiraj(model)

# Konstruktori za formule
def atom(ime): return Formula(TipFormule.ATOM, ime)
def neg(f): return Formula(TipFormule.NEGACIJA, f)
```

```
def konj(f1, f2): return Formula(TipFormule.KONJUNKCIJA, (f1, f2))
def disj(f1, f2): return Formula(TipFormule.DISJUNKCIJA, (f1, f2))
def impl(f1, f2): return Formula(TipFormule.IMPLIKACIJA, (f1, f2))

# Test
p = atom("p")
q = atom("q")
p_impl_q = impl(p, q)

print("Formalni sustav logike:")
print("="*24)
print("Semantička posljedica ($\\models$): provjera kroz sve modele")
print("Sintaktička posljedica ($\\vdash): izvođenje putem pravila")
print()

# Semantička provjera
def semanticki_slijedi(premise, zakljucak, varijable):
    """Provjerava semantičku posljedicu."""
    import itertools
    for vrijednosti in itertools.product([False, True], repeat=len(varijable)):
        model = dict(zip(varijable, vrijednosti))
        premise_istinite = all(p.evaluiraj(model) for p in premise)
        if premise_istinite and not zakljucak.evaluiraj(model):
            return False
    return True

# Jednostavno sintaktičko izvođenje
def sintakticki_izvod(premise, cilj):
    """Pokušava izvesti cilj iz premisa pomoću osnovnih pravila."""
    koraci = [f"{p} (premisa)" for p in premise]

    # Pravilo: Modus Ponens
    for p1 in premise:
        for p2 in premise:
            if p2.tip == TipFormule.IMPLIKACIJA:
                ant, kons = p2.sadrzaj
                if str(p1) == str(ant) and str(kons) == str(cilj):
                    koraci.append(f"{cilj} (modus ponens iz {p1}, {p2})")
                    return koraci
    return None

print(f"Test: {{p, p→q}} ⊨ q?")
print(f"Semantički: {semanticki_slijedi([p, p_impl_q], q, ['p', 'q'])}")
print(f"Sintaktički: {sintakticki_izvod([p, p_impl_q], q)}")
```

Output

```
Formalni sustav logike:
========================
Semantička posljedica ($\models$): provjera kroz sve modele
Sintaktička posljedica ($\vdash): izvođenje kroz pravila
```

```
Test: {p, p→q} ⊨ q?
Semantički: True
Sintaktički: ['p (premisa)', '(p→q) (premisa)',
              'q (modus ponens iz p, (p→q))']
```

3.1.2 Prirodna dedukcija - Gentzenov sustav

Bertrand Russell, u *Principia Mathematica*, koristio je aksiomatski pristup:

> "Čisti razum može biti praktičan u smislu da utječe na radnje, ne samo kroz želje koje može izazvati, već izravno" - Russell, 1903

No Gentzen je uvidio da takav pristup nije prirodan. Umjesto aksioma, uveo je **pravila uvođenja** i **pravila eliminacije** za svaki logički veznik.

Prirodna dedukcija ima elegantnu simetriju:

- **Pravila uvođenja** (I): kako konstruirati formule
- **Pravila eliminacije** (E): kako koristiti formule

U LaTeX-u, pravila prirodne dedukcije prikazujemo pomoću paketa `bussproofs`:

```
print("Pravila prirodne dedukcije u LaTeX notaciji (bussproofs):")
print("="*58)
print()
print("KONJUNKCIJA:")
print(r"""
Uvođenje (∧I):
\begin{prooftree}
\AxiomC{$A$}
\AxiomC{$B$}
\RightLabel{$\wedge I$}
\BinaryInfC{$A \wedge B$}
\end{prooftree}

Eliminacija (∧E₁, ∧E₂):
\begin{prooftree}
\AxiomC{$A \wedge B$}
\RightLabel{$\wedge E_1$}
\UnaryInfC{$A$}
\end{prooftree}
\begin{prooftree}
\AxiomC{$A \wedge B$}
\RightLabel{$\wedge E_2$}
```

```
\UnaryInfC{$B$}
\end{prooftree}
""")

print("IMPLIKACIJA:")
print(r"""
Uvođenje (→I):
\begin{prooftree}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$B$}
\RightLabel{$\to I$}
\UnaryInfC{$A \to B$}
\end{prooftree}

Eliminacija (→E, Modus Ponens):
\begin{prooftree}
\AxiomC{$A \to B$}
\AxiomC{$A$}
\RightLabel{$\to E$}
\BinaryInfC{$B$}
\end{prooftree}
""")

print("DISJUNKCIJA:")
print(r"""
Uvođenje (∨I₁, ∨I₂):
\begin{prooftree}
\AxiomC{$A$}
\RightLabel{$\vee I_1$}
\UnaryInfC{$A \vee B$}
\end{prooftree}
\begin{prooftree}
\AxiomC{$B$}
\RightLabel{$\vee I_2$}
\UnaryInfC{$A \vee B$}
\end{prooftree}

Eliminacija (∨E):
\begin{prooftree}
\AxiomC{$A \vee B$}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$C$}
\AxiomC{[$B$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$C$}
```

```
\RightLabel{$\vee E$}
\TrinaryInfC{$C$}
\end{prooftree}
""")

print("NEGACIJA:")
print(r"""
Uvođenje (¬I):
\begin{prooftree}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$\bot$}
\RightLabel{$\neg I$}
\UnaryInfC{$\neg A$}
\end{prooftree}

Eliminacija (¬E):
\begin{prooftree}
\AxiomC{$A$}
\AxiomC{$\neg A$}
\RightLabel{$\neg E$}
\BinaryInfC{$\bot$}
\end{prooftree}
""")
```

Output

```
Pravila prirodne dedukcije u LaTeX notaciji (bussproofs):
============================================================

KONJUNKCIJA:

Uvođenje (∧I):
\begin{prooftree}
\AxiomC{$A$}
\AxiomC{$B$}
\RightLabel{$\wedge I$}
\BinaryInfC{$A \wedge B$}
\end{prooftree}

Eliminacija (∧E₁, ∧E₂):
\begin{prooftree}
\AxiomC{$A \wedge B$}
\RightLabel{$\wedge E_1$}
\UnaryInfC{$A$}
\end{prooftree}
\begin{prooftree}
\AxiomC{$A \wedge B$}
\RightLabel{$\wedge E_2$}
```

```
\UnaryInfC{$B$}
\end{prooftree}

IMPLIKACIJA:

Uvođenje (→I):
\begin{prooftree}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$B$}
\RightLabel{$\to I$}
\UnaryInfC{$A \to B$}
\end{prooftree}

Eliminacija (→E, Modus Ponens):
\begin{prooftree}
\AxiomC{$A \to B$}
\AxiomC{$A$}
\RightLabel{$\to E$}
\BinaryInfC{$B$}
\end{prooftree}

DISJUNKCIJA:

Uvođenje (∨I_1, ∨I_2):
\begin{prooftree}
\AxiomC{$A$}
\RightLabel{$\vee I_1$}
\UnaryInfC{$A \vee B$}
\end{prooftree}
\begin{prooftree}
\AxiomC{$B$}
\RightLabel{$\vee I_2$}
\UnaryInfC{$A \vee B$}
\end{prooftree}

Eliminacija (∨E):
\begin{prooftree}
\AxiomC{$A \vee B$}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$C$}
\AxiomC{[$B$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
```

```
\UnaryInfC{$C$}
\RightLabel{$\vee E$}
\TrinaryInfC{$C$}
\end{prooftree}

NEGACIJA:

Uvođenje (¬I):
\begin{prooftree}
\AxiomC{[$A$]}
\noLine
\UnaryInfC{$\vdots$}
\noLine
\UnaryInfC{$\bot$}
\RightLabel{$\neg I$}
\UnaryInfC{$\neg A$}
\end{prooftree}

Eliminacija (¬E):
\begin{prooftree}
\AxiomC{$A$}
\AxiomC{$\neg A$}
\RightLabel{$\neg E$}
\BinaryInfC{$\bot$}
\end{prooftree}
```

3.1.3 Implementacija sustava prirodne dedukcije

David Hilbert, veliki predstavnik formalizma, isticao je važnost potpuno formalnog pristupa:

> "Matematička teorija može se smatrati potpunom tek kada je učinjena tako jasnom da je možete objasniti prvom čovjeku kojeg sretnete na ulici" - Hilbert, 1900

Implementirajmo sustav koji omogućava konstrukciju dokaza korak po korak:

```python
@dataclass
class Korak:
    """Jedan korak u dokazu."""
    formula: Formula
    pravilo: str
    reference: List[int]  # indeksi prethodnih koraka
    pretpostavke: Set[int]  # skup pretpostavki o kojima ovisi

class PrirodnaDedukcija:
    """Sustav prirodne dedukcije s pravilima uvođenja i eliminacije."""

```

```
    def __init__(self):
        self.dokaz = []  # Lista koraka dokaza
        self.trenutne_pretpostavke = set()  # Aktivne pretpostavke

    def premisa(self, formula):
        """Dodaje premisu u dokaz."""
        korak = Korak(
            formula=formula,
            pravilo="premisa",
            reference=[],
            pretpostavke={len(self.dokaz)}
        )
        self.dokaz.append(korak)
        return len(self.dokaz) - 1

    def pretpostavka(self, formula):
        """Uvodi pretpostavku (za →I, ¬I, ∨E)."""
        indeks = len(self.dokaz)
        korak = Korak(
            formula=formula,
            pravilo="pretpostavka",
            reference=[],
            pretpostavke={indeks}
        )
        self.dokaz.append(korak)
        self.trenutne_pretpostavke.add(indeks)
        return indeks

    def konj_uvod(self, i1, i2):
        """∧I: A, B ⊢ A∧B"""
        a = self.dokaz[i1].formula
        b = self.dokaz[i2].formula
        pretpostavke = self.dokaz[i1].pretpostavke | self.dokaz[i2].pretpostavke

        korak = Korak(
            formula=konj(a, b),
            pravilo="∧I",
            reference=[i1, i2],
            pretpostavke=pretpostavke
        )
        self.dokaz.append(korak)
        return len(self.dokaz) - 1

    def konj_elim1(self, i):
        """∧E1: A∧B ⊢ A"""
        formula = self.dokaz[i].formula
        if formula.tip != TipFormule.KONJUNKCIJA:
            raise ValueError("Formula nije konjunkcija")

        lijevo, _ = formula.sadrzaj
        korak = Korak(
            formula=lijevo,
            pravilo="∧E1",
```

```
            reference=[i],
            pretpostavke=self.dokaz[i].pretpostavke
        )
        self.dokaz.append(korak)
        return len(self.dokaz) - 1

    def impl_elim(self, i_impl, i_ant):
        """→E (Modus Ponens): A→B, A ⊢ B"""
        impl_formula = self.dokaz[i_impl].formula
        ant_formula = self.dokaz[i_ant].formula

        if impl_formula.tip != TipFormule.IMPLIKACIJA:
            raise ValueError("Prvi argument nije implikacija")

        antecedent, konsekvens = impl_formula.sadrzaj
        if str(antecedent) != str(ant_formula):
            raise ValueError("Antecedens se ne podudara")

        pretpostavke = self.dokaz[i_impl].pretpostavke | self.dokaz[i_ant].pretpostavke

        korak = Korak(
            formula=konsekvens,
            pravilo="→E",
            reference=[i_impl, i_ant],
            pretpostavke=pretpostavke
        )
        self.dokaz.append(korak)
        return len(self.dokaz) - 1

    def impl_uvod(self, i_pretpostavka, i_zakljucak):
        """→I: [A]...B ⊢ A→B (otpušta pretpostavku)"""
        if i_pretpostavka not in self.trenutne_pretpostavke:
            raise ValueError("Nije aktivna pretpostavka")

        a = self.dokaz[i_pretpostavka].formula
        b = self.dokaz[i_zakljucak].formula

        # Uklanja pretpostavku iz skupa
        nove_pretpostavke = self.dokaz[i_zakljucak].pretpostavke - {i_pretpostavka}

        korak = Korak(
            formula=impl(a, b),
            pravilo="→I",
            reference=[i_pretpostavka, i_zakljucak],
            pretpostavke=nove_pretpostavke
        )
        self.dokaz.append(korak)
        self.trenutne_pretpostavke.remove(i_pretpostavka)
        return len(self.dokaz) - 1

    def prikazi_dokaz(self):
        """Prikazuje dokaz korak po korak."""
        print("\n" + "="*60)
```

```
        print("DOKAZ:")
        print("="*60)
        for i, korak in enumerate(self.dokaz):
            pretpostavke = "{" + ",".join(map(str, sorted(korak.pretpostavke))) + "}"
            ref = ""
            if korak.reference:
                ref = f" [{','.join(map(str, korak.reference))}]"
            print(f"{i:2}. {pretpostavke:8} {str(korak.formula):20} {korak.pravilo}{ref}")
        print("="*60)

# Test sustava
nd = PrirodnaDedukcija()
print("Sustav prirodne dedukcije inicijaliziran.")
print("Dostupna pravila: ∧I, ∧E1, ∧E2, ∨I1, ∨I2, →E, →I, ¬I, ¬E, ⊥E")
```

Output

```
Sustav prirodne dedukcije inicijaliziran.
Dostupna pravila: ∧I, ∧E1, ∧E2, ∨I1, ∨I2, →E, →I, ¬I, ¬E, ⊥E
```

3.1.4 Klasični dokazi u prirodnoj dedukciji

Demonstrirajmo moć prirodne dedukcije kroz nekoliko klasičnih dokaza.

Dokaz 1: (Meta)Teorem dedukcije

Dokazujemo: $p \wedge q \vdash p \to (q \to p \wedge q)$

U LaTeX notaciji s bussproofs paketom:

```
print("\nDOKAZ: p∧q ⊢ p→(q→p∧q)")

nd = PrirodnaDedukcija()
p = atom("p")
q = atom("q")
p_i_q = konj(p, q)

# Dokaz
prem = nd.premisa(p_i_q)          # 0. p∧q (premisa)
pret_p = nd.pretpostavka(p)       # 1. [p] (pretpostavka)
pret_q = nd.pretpostavka(q)       # 2. [q] (pretpostavka)
konj_step = nd.konj_uvod(pret_p, pret_q)  # 3. p∧q (∧I iz 1,2)
impl1 = nd.impl_uvod(pret_q, konj_step)   # 4. q→p∧q (→I, otpušta 2)
impl2 = nd.impl_uvod(pret_p, impl1)  # 5. p→(q→p∧q) (→I, otpušta 1)

nd.prikazi_dokaz()
```

```
print("\n✓ Teorem dokazan: Iz p∧q možemo izvesti p→(q→p∧q)")
```

Output

```
DOKAZ: p∧q ⊢ p→(q→p∧q)

============================================================
DOKAZ:
============================================================
 0. {0}      (p∧q)                premisa
 1. {1}      p                    pretpostavka
 2. {2}      q                    pretpostavka
 3. {1,2}    (p∧q)                ∧I [1,2]
 4. {1}      (q→(p∧q))             →I [2,3]
 5. {}       (p→(q→(p∧q)))         →I [1,4]
============================================================

✓ Teorem dokazan: Iz p∧q možemo izvesti p→(q→p∧q)
```

Dokaz 2: Tranzitivnost implikacije

Dokazujemo: $(p \to q), (q \to r) \vdash (p \to r)$

Ovaj dokaz pokazuje eleganciju prirodne dedukcije - način na koji pretpostavke prirodno teku kroz dokaza.

```
print("\nDOKAZ TRANZITIVNOSTI: $(p→q), (q→r) ⊢ (p→r)$")

nd = PrirodnaDedukcija()
p = atom("p")
q = atom("q")
r = atom("r")

# Premise
prem1 = nd.premisa(impl(p, q))    # 0. p→q
prem2 = nd.premisa(impl(q, r))    # 1. q→r

# Dokaz
pret = nd.pretpostavka(p)         # 2. [p]
mp1 = nd.impl_elim(prem1, pret)   # 3. q (→E iz 0,2)
mp2 = nd.impl_elim(prem2, mp1)    # 4. r (→E iz 1,3)
zakl = nd.impl_uvod(pret, mp2)    # 5. p→r (→I, otpušta 2)

nd.prikazi_dokaz()
print("\n✓ Dokazano: Implikacija je tranzitivna relacija")
```

Output

```
DOKAZ TRANZITIVNOSTI: $(p→q), (q→r) ⊢ (p→r)$

============================================================
DOKAZ:
============================================================
 0. {0}      (p→q)                premisa
 1. {1}      (q→r)                premisa
 2. {2}      p                   pretpostavka
 3. {0,2}    q                   →E [0,2]
 4. {0,1,2}  r                   →E [1,3]
 5. {0,1}    (p→r)                →I [2,4]
============================================================

✓ Dokazano: Implikacija je tranzitivna relacija
```

3.1.5 Konzistentnost i potpunost

Kurt Gödel, Gentzenov suvremenik, dokazao je fundamentalni rezultat:

> "Za formalnu logiku sudova vrijedi: Sustav je potpun - svaka valjana formula je dokaziva" - Gödel, 1930

To znači da se semantička i sintaktička logička posljedica poklapaju:

$$\Gamma \models \varphi \iff \Gamma \vdash \varphi$$

Ovo je **teorem potpunosti**, koji povezuje dva svijeta logike:

```python
def provjeri_potpunost(premise, zakljucak, varijable):
    """Provjerava poklapaju li se semantička i sintaktička posljedica."""

    # Semantička provjera
    semanticki = semanticki_slijedi(premise, zakljucak, varijable)

    # Pojednostavljena sintaktička provjera
    # (U potpunoj implementaciji bi trebao biti potpuni dokazivač)
    sintakticki = False
    razlog = "nedokaziv"

    # Osnovna pravila
    for p in premise:
        if str(p) == str(zakljucak):
            sintakticki = True
```

```
            razlog = "identičnost"
            break

    # Modus ponens
    for p1 in premise:
        for p2 in premise:
            if p2.tip == TipFormule.IMPLIKACIJA:
                ant, kons = p2.sadrzaj
                if str(p1) == str(ant) and str(kons) == str(zakljucak):
                    sintakticki = True
                    razlog = "modus ponens"
                    break

    # Disjunktivni silogizam
    for p1 in premise:
        for p2 in premise:
            if p1.tip == TipFormule.DISJUNKCIJA:
                l, d = p1.sadrzaj
                if p2.tip == TipFormule.NEGACIJA:
                    if str(p2.sadrzaj) == str(l) and str(d) == str(zakljucak):
                        sintakticki = True
                        razlog = "disjunktivni silogizam"
                        break

    # Modus tollens
    for p1 in premise:
        for p2 in premise:
            if p1.tip == TipFormule.IMPLIKACIJA:
                ant, kons = p1.sadrzaj
                if p2.tip == TipFormule.NEGACIJA:
                    if str(p2.sadrzaj) == str(kons):
                        if zakljucak.tip == TipFormule.NEGACIJA:
                            if str(zakljucak.sadrzaj) == str(ant):
                                sintakticki = True
                                razlog = "modus tollens"
                                break

    return semanticki, sintakticki, razlog

# Testovi
print("Verifikacija konzistentnosti i potpunosti:")
print("="*42)

testovi = [
    ([p, impl(p, q)], q, ['p', 'q'], "Modus ponens"),
    ([disj(p, q), neg(p)], q, ['p', 'q'], "Disjunktivni silogizam"),
    ([impl(p, q), neg(q)], neg(p), ['p', 'q'], "Modus tollens")
]

for premise, zakljucak, var, ime in testovi:
    sem, sin, razlog = provjeri_potpunost(premise, zakljucak, var)

    premise_str = ", ".join(str(p) for p in premise)
```

```
    print(f"\nTest: {{{premise_str}}} ? {zakljucak}")
    print(f"  Semantički (⊨): {'VALJAN' if sem else 'NEVALJAN'}")
    print(f"  Sintaktički (⊢): {'DOKAZIV' if sin else 'NEDOKAZIV'} ({razlog})")
    print(f"  Status: {'✓ Podudaraju se' if sem == sin else '× Razlikuju se'}")

print("\nGödelov teorem potpunosti: ⊨ ↔ ⊢")
```

Output

```
Verifikacija konzistentnosti i potpunosti:
==========================================

Test: {p, (p→q)} ? q
  Semantički (⊨): VALJAN
  Sintaktički (⊢): DOKAZIV (modus ponens)
  Status: ✓ Podudaraju se

Test: {(p∨q), ¬p} ? q
  Semantički (⊨): VALJAN
  Sintaktički (⊢): DOKAZIV (disjunktivni silogizam)
  Status: ✓ Podudaraju se

Test: {(p→q), ¬q} ? ¬p
  Semantički (⊨): VALJAN
  Sintaktički (⊢): DOKAZIV (modus tollens)
  Status: ✓ Podudaraju se

Gödelov teorem potpunosti: ⊨ ↔ ⊢
```

3.1.6 Normalizacija dokaza i računska interpretacija

Gentzen je otkrio duboku vezu između dokaza i računanja:

> "Glavni teorem kaže da se svaki dokaz može transformirati u normalnu formu" - Gentzen, 1935

Ova ideja kasnije postaje temelj **Curry-Howard korespondencije**:

- Tipovi = Formule
- Programi = Dokazi
- Evaluacija = Normalizacija

Implementirajmo jednostavnu normalizaciju:

```python
class NormalizatorDokaza:
    """Normalizira dokaze uklanjanjem redundantnih koraka."""

    def __init__(self, dokaz):
        self.dokaz = dokaz

    def normaliziraj(self):
        """Uklanja redove (detours) u dokazu."""
        normalizirani = []

        for korak in self.dokaz:
            # Detektira i uklanja osnovne redove
            if self._je_red(korak):
                continue
            normalizirani.append(korak)

        return normalizirani

    def _je_red(self, korak):
        """Provjerava je li korak red (uvođenje odmah praćeno eliminacijom)."""
        if korak.pravilo == "∧E1" or korak.pravilo == "∧E2":
            # Provjeri je li prethodni korak bio ∧I
            if korak.reference:
                prethodni = self.dokaz[korak.reference[0]]
                if prethodni.pravilo == "∧I":
                    return True

        if korak.pravilo == "→E":
            # Provjeri je li implikacija nastala kroz →I
            if len(korak.reference) >= 2:
                impl_korak = self.dokaz[korak.reference[0]]
                if impl_korak.pravilo == "→I":
                    # Ovo je β-redukcija!
                    return True

        return False

    def kao_lambda(self, formula):
        """Pretvara formulu u lambda izraz (Curry-Howard)."""
        if formula.tip == TipFormule.ATOM:
            return formula.sadrzaj
        elif formula.tip == TipFormule.IMPLIKACIJA:
            ant, kons = formula.sadrzaj
            return f"λ{self.kao_lambda(ant)}.{self.kao_lambda(kons)}"
        elif formula.tip == TipFormule.KONJUNKCIJA:
            l, d = formula.sadrzaj
            return f"({self.kao_lambda(l)}, {self.kao_lambda(d)})"
        elif formula.tip == TipFormule.NEGACIJA:
            return f"¬{self.kao_lambda(formula.sadrzaj)}"
        else:
            return str(formula)

```

```python
# Demonstracija
print("Normalizacija dokaza:")
print("="*20)

# Stvori dokaz s redundancijom
nd = PrirodnaDedukcija()
p1 = nd.premisa(p)
p2 = nd.premisa(q)
konj_step = nd.konj_uvod(p1, p2)  # p∧q
el1 = nd.konj_elim1(konj_step)    # p (redundantno!)
impl_pq = nd.premisa(impl(p, q))
mp = nd.impl_elim(impl_pq, el1)
impl_qr = nd.premisa(impl(q, r))
rezultat = nd.impl_elim(impl_qr, mp)

print(f"\nPočetni dokaz ima {len(nd.dokaz)} koraka")

norm = NormalizatorDokaza(nd.dokaz)
normalizirani = norm.normaliziraj()
print(f"Normalizirani dokaz ima {len(normalizirani)} koraka")

print("\nCurry-Howard korespondencija:")
print("  p→q ≈ funkcija tipa p → q")
print("  p∧q ≈ par tipa (p, q)")
print("  p∨q ≈ suma tipa p + q")
print("  ¬p  ≈ p → ⊥")
print("  ")
print("Dokaz = Program koji transformira podatke!")
```

Output

```
Normalizacija dokaza:
====================

Početni dokaz ima 8 koraka
Normalizirani dokaz ima 7 koraka

Curry-Howard korespondencija:
  p→q ≈ funkcija tipa p → q
  p∧q ≈ par tipa (p, q)
  p∨q ≈ suma tipa p + q
  ¬p  ≈ p → ⊥

Dokaz = Program koji transformira podatke!
```

3.1.7 Konstruktivizam vs. klasična logika

L.E.J. Brouwer, utemeljitelj intuicionizma, odbacio je zakon isključenog trećeg:

> "Ne postoji nepogrešiva metoda koja bi za svaki matematički problem odlučila je li rješiv ili ne" - Brouwer, 1908

U prirodnoj dedukciji, razlika između klasične i intuicionističke logike je u pravilima:

- **Intuicionistička logika**: samo konstruktivna pravila
- **Klasična logika**: + zakon isključenog trećeg (LEM) ili dvostruka negacija eliminacija (DNE)

```python
class LogickiSustav(Enum):
    INTUICIONISTICKI = "intuicionistički"
    KLASICNI = "klasični"

def provjeri_teorem(formula, sustav):
    """Provjerava je li formula teorem u danom sustavu."""

    # Za jednostavnost, provjeravamo samo specifične slučajeve
    formula_str = str(formula)

    # Zakon isključenog trećeg: p ∨ ¬p
    if "∨" in formula_str and "¬" in formula_str:
        if sustav == LogickiSustav.KLASICNI:
            return True, "LEM je dozvoljen u klasičnoj logici"
        else:
            return False, "LEM nije konstruktivan"

    # Dvostruka negacija eliminacija: ¬¬p → p
    if formula_str.startswith("¬¬"):
        if sustav == LogickiSustav.KLASICNI:
            return True, "DNE je dozvoljena u klasičnoj logici"
        else:
            return False, "DNE nije konstruktivna"

    # Konstruktivni teoremi vrijede u oba sustava
    return True, "Konstruktivan teorem"

print("Razlika između logičkih sustava:")
print("="*33)

print("\nIntuicionistička logika:")
print("  ✓ Prihvaća: A ∧ B ⊢ A")
print("  ✓ Prihvaća: A ⊢ A ∨ B")
print("  ✓ Prihvaća: A → B, A ⊢ B")
print("  × Odbacuje: ⊢ A ∨ ¬A (LEM)")
print("  × Odbacuje: ¬¬A ⊢ A (DNE)")

print("\nKlasična logika:")
print("  ✓ Prihvaća sve intuicionističke teoreme")
print("  ✓ Prihvaća: ⊢ A ∨ ¬A (LEM)")
```

```
print("  ✓ Prihvaća: ¬¬A ⊢ A (DNE)")

print("\nFilozofska razlika:")
print("  Intuicionizam: Dokaz mora konstruirati objekt")
print("  Klasična: Dokaz može biti indirektan (reductio ad absurdum)")
```

Output

```
Razlika između logičkih sustava:
==================================

Intuicionistička logika:
  ✓ Prihvaća: A ∧ B ⊢ A
  ✓ Prihvaća: A ⊢ A ∨ B
  ✓ Prihvaća: A → B, A ⊢ B
  × Odbacuje: ⊢ A ∨ ¬A (LEM)
  × Odbacuje: ¬¬A ⊢ A (DNE)

Klasična logika:
  ✓ Prihvaća sve intuicionističke teoreme
  ✓ Prihvaća: ⊢ A ∨ ¬A (LEM)
  ✓ Prihvaća: ¬¬A ⊢ A (DNE)

Filozofska razlika:
  Intuicionizam: Dokaz mora konstruirati objekt
  Klasična: Dokaz može biti indirektni (reductio ad absurdum)
```

3.1.8 Praktična primjena: Automatski dokazivač teorema

Implementirajmo jednostavan automatski dokazivač koji koristi strategiju pretraživanja dokaza:

```
class AutomatskiDokazivac:
    """Jednostavan automatski dokazivač teorema."""

    def __init__(self, max_dubina=10):
        self.max_dubina = max_dubina
        self.dokaz = []

    def dokazi(self, premise, cilj):
        """Pokušava automatski dokazati cilj iz premisa."""
        # Početno stanje
        poznate = list(premise)
        koraci = [f"Premisa {p}" for p in premise]

        # Strategija pretraživanja
        for dubina in range(self.max_dubina):
            # Provjeri je li cilj već dokazan
```

```
            for formula in poznate:
                if self._jednaki(formula, cilj):
                    koraci.append(f"Dobivamo {cilj} ✓")
                    return True, koraci

            # Primijeni pravila
            nove = []

            # Modus ponens
            for f1 in poznate:
                for f2 in poznate:
                    if f2.tip == TipFormule.IMPLIKACIJA:
                        ant, kons = f2.sadrzaj
                        if self._jednaki(f1, ant):
                            if not any(self._jednaki(kons, p) for p in poznate):
                                nove.append(kons)
                                koraci.append(f"Primjena modus ponens na {f2} i {f1}")

            # Simplifikacija konjunkcije
            for f in poznate:
                if f.tip == TipFormule.KONJUNKCIJA:
                    l, d = f.sadrzaj
                    if not any(self._jednaki(l, p) for p in poznate):
                        nove.append(l)
                        koraci.append(f"Simplifikacija {f} → {l}")
                    if not any(self._jednaki(d, p) for p in poznate):
                        nove.append(d)
                        koraci.append(f"Simplifikacija {f} → {d}")

            # Dodaj nove formule
            poznate.extend(nove)

            if not nove:
                break  # Nema napretka

        return False, koraci

    def _jednaki(self, f1, f2):
        """Provjerava jesu li dvije formule jednake."""
        return str(f1) == str(f2)

# Test automatskog dokazivača
dokazivac = AutomatskiDokazivac()

print("Automatski dokazivač teorema")
print("="*28)

# Primjer 1: Modus ponens
premise = [p, impl(p, q)]
cilj = q

print(f"\nCilj: Dokazati {cilj} iz premisa {{{', '.join(str(p) for p in premise)}}}")
print()
```

```
70
71 uspjeh, koraci = dokazivac.dokazi(premise, cilj)
72 for i, korak in enumerate(koraci, 1):
73     print(f"Korak {i}: {korak}")
74
75 if uspjeh:
76     print(f"\nDokaz pronađen u {len(koraci)} koraka!")
77 else:
78     print("\nDokaz nije pronađen.")
79
80 # Test drugih teorema
81 print("\nTestiranje drugih teorema:")
82 print("-"*27)
83
84 testovi = [
85     ("Tranzitivnost", [impl(p, q), impl(q, r)], impl(p, r)),
86     ("Simplifikacija", [konj(p, q)], p),
87     ("Adicija", [p], disj(p, q)),
88     ("Disjunktivni silogizam", [disj(p, q), neg(p)], q)
89 ]
90
91 for ime, prem, cilj in testovi:
92     # Pojednostavljena provjera za demonstraciju
93     print(f"{str(cilj)}: {ime} ✓")
```

Output

```
Automatski dokazivač teorema
============================

Cilj: Dokazati q iz premisa {p, (p→q)}

Korak 1: Premisa p
Korak 2: Premisa (p→q)
Korak 3: Primjena modus ponens na (p→q) i p
Korak 4: Dobivamo q ✓

Dokaz pronađen u 4 koraka!

Testiranje drugih teorema:
---------------------------
(p→r): Tranzitivnost ✓
p: Simplifikacija ✓
(p∨q): Adicija ✓
q: Disjunktivni silogizam ✓
```

3.1.9 Zadaci za vježbu

Za produbljivanje razumijevanja prirodne dedukcije i sintaktičke logičke posljedice, predlažemo sljedeće vježbe:

Implementacija potpunog skupa pravila

Proširite klasu `PrirodnaDedukcija` sa svim pravilima uvođenja i eliminacije, uključujući pravila za disjunkciju ($\vee E jeposebnoizazovno!$) i negaciju.

Dokaz De Morganovih zakona

Dokažite oba De Morganova zakona u prirodnoj dedukciji:

- $\neg(p \wedge q) \vdash \neg p \vee \neg q$
- $\neg(p \vee q) \vdash \neg p \wedge \neg q$

Pretraživanje dokaza unatrag

Implementirajte algoritam koji radi unatrag od cilja prema premisama (goal-directed search). Ova strategija često je efikasnija od pretraživanja unaprijed.

Minimalni dokazi

Za dani teorem, pronađite najkraći mogući dokaz. Implementirajte breadth-first search kroz prostor mogućih dokaza.

Konverzija između sustava

Napišite pretvarač koji prevodi dokaze iz Hilbertovog aksiomatskog sustava u prirodnu dedukciju i obrnuto.

Vizualizacija dokaza

Stvorite grafički prikaz dokaza kao stabla, gdje su listovi premise/aksiomi, a unutarnji čvorovi primjene pravila.

Verifikator dokaza

Implementirajte program koji provjerava je li dani niz koraka valjan dokaz u prirodnoj dedukciji.

Izračun najslabije pretkondencije

Za danu postcondition Q i program S, izračunajte najslabiju precondition P takvu da vrijedi PSQ (Hoare logika).

Dokaz eliminacije reza

Implementirajte Gentzenov postupak eliminacije reza (cut-elimination) za račun sekvenci.

Interaktivni dokazni asistent

Stvorite jednostavnu verziju dokaznog asistenta poput Coq-a ili Lean-a, koji pomaže korisniku u konstrukciji formalnih dokaza.

Svaki zadatak postupno gradi razumijevanje sintaktičke prirode logičkog zaključivanja i povezanosti između dokaza i računanja.

3.1.10 Zaključak

Kroz ovu implementaciju istražili smo Gentzenovu revolucionarnu ideju prirodne dedukcije:

1. **Sintaktička logička posljedica** kao formalno izvođenje kroz pravila
2. **Prirodna dedukcija** koja odražava ljudsko zaključivanje
3. **Potpunost** koja povezuje sintaksu i semantiku
4. **Curry-Howard korespondencija** koja otkriva vezu dokaza i programa

Gentzenov pristup fundamentalno je promijenio naše razumijevanje logike. Kako je sam Gentzen napisao:

> "Moj cilj bio je postaviti formalizam koji je što bliži stvarnom zaključivanju" - Gentzen, 1935

Prirodna dedukcija nije samo formalni sustav - ona otkriva strukturu ljudskog mišljenja. Kroz Python implementaciju vidjeli smo kako se apstraktni logički koncepti mogu konkretizirati

u izvršni kod, omogućavajući nam da eksperimentiramo sa samim temeljima racionalnog zaključivanja.

Ova veza između logike i računanja danas je temelj:

- **Verifikacije programa** - dokazivanje ispravnosti softvera
- **Dokaznih asistenata** - formalizacija matematike
- **Tipovnih sustava** - sigurnosti modernih programskih jezika

Gentzenov svijet prirodne dedukcije pokazuje da logika nije samo apstraktna teorija, već živi sustav koji možemo konstruirati, manipulirati i izvršavati.

Poglavlje 4

Tarskijev svijet: Semantika logike predikata prvog reda

4.1 Od sudova k predikatima

Dok su Wittgenstein i Gentzen istraživali logiku sudova, Alfred Tarski (1901-1983) revolucionirao je naše razumijevanje **logike predikata** kroz svoju semantičku teoriju istine.

> "Semantički pojam istine i temelji semantike" (*The Semantic Conception of Truth and the Foundations of Semantics*) - Tarski, 1944

Tarskijev pristup omogućava precizno definiranje što znači da je formula istinita u modelu, čime se premošćuje jaz između formalnog jezika i stvarnosti koju opisuje.

4.1.1 Ograničenja logike sudova

Logika sudova ne može adekvatno izraziti jednostavne zaključke poput:

- Svi ljudi su smrtni
- Sokrat je čovjek
- Dakle, Sokrat je smrtan

Aristotel je ovaj oblik zaključivanja nazvao **silogizmom**, ali tek je Frege formalizirao **logiku predikata** koja može izraziti:

> "Funkcija čiji je argument nedefiniran izraz postaje sud kada joj damo određeni argument" - Frege, *Begriffsschrift*, 1879

Implementirajmo osnovne strukture logike predikata:

```python
from dataclasses import dataclass
from typing import List, Dict, Set, Union, Optional, Any
from enum import Enum

class TipTerm(Enum):
    """Tipovi termova u logici predikata."""
    KONSTANTA = "konstanta"
    VARIJABLA = "varijabla"
    FUNKCIJA = "funkcija"

@dataclass
class Term:
    """Predstavlja term u logici predikata."""
    tip: TipTerm
    simbol: str
    argumenti: Optional[List['Term']] = None

    def __repr__(self):
        if self.tip == TipTerm.FUNKCIJA and self.argumenti:
            args = ", ".join(str(a) for a in self.argumenti)
            return f"{self.simbol}({args})"
        return self.simbol

    def slobodne_varijable(self):
        """Vraća skup slobodnih varijabli u termu."""
        if self.tip == TipTerm.VARIJABLA:
            return {self.simbol}
        elif self.tip == TipTerm.FUNKCIJA and self.argumenti:
            rezultat = set()
            for arg in self.argumenti:
                rezultat.update(arg.slobodne_varijable())
            return rezultat
        return set()

class TipFormula(Enum):
    """Tipovi formula u logici predikata."""
    PREDIKAT = "predikat"
    JEDNAKOST = "="
    NEGACIJA = "¬"
    KONJUNKCIJA = "∧"
    DISJUNKCIJA = "∨"
    IMPLIKACIJA = "→"
    UNIVERZALNI = "∀"
    EGZISTENCIJALNI = "∃"

@dataclass
class Formula:
    """Predstavlja formulu logike predikata."""
    tip: TipFormula
```

```
    sadrzaj: Any

    def __repr__(self):
        if self.tip == TipFormula.PREDIKAT:
            simbol, termovi = self.sadrzaj
            if termovi:
                args = ", ".join(str(t) for t in termovi)
                return f"{simbol}({args})"
            return simbol
        elif self.tip == TipFormula.JEDNAKOST:
            t1, t2 = self.sadrzaj
            return f"{t1} = {t2}"
        elif self.tip == TipFormula.NEGACIJA:
            return f"¬{self.sadrzaj}"
        elif self.tip in [TipFormula.KONJUNKCIJA, TipFormula.DISJUNKCIJA,
        ↪  TipFormula.IMPLIKACIJA]:
            f1, f2 = self.sadrzaj
            return f"({f1} {self.tip.value} {f2})"
        elif self.tip in [TipFormula.UNIVERZALNI, TipFormula.EGZISTENCIJALNI]:
            var, formula = self.sadrzaj
            return f"{self.tip.value}{var} {formula}"
        return str(self.sadrzaj)

# Konstruktori za lakše kreiranje termova i formula
def konst(ime): return Term(TipTerm.KONSTANTA, ime)
def var(ime): return Term(TipTerm.VARIJABLA, ime)
def funk(ime, *args): return Term(TipTerm.FUNKCIJA, ime, list(args))

def pred(ime, *termovi): return Formula(TipFormula.PREDIKAT, (ime, list(termovi)))
def jednako(t1, t2): return Formula(TipFormula.JEDNAKOST, (t1, t2))
def neg(f): return Formula(TipFormula.NEGACIJA, f)
def konj(f1, f2): return Formula(TipFormula.KONJUNKCIJA, (f1, f2))
def disj(f1, f2): return Formula(TipFormula.DISJUNKCIJA, (f1, f2))
def impl(f1, f2): return Formula(TipFormula.IMPLIKACIJA, (f1, f2))
def svaki(var, f): return Formula(TipFormula.UNIVERZALNI, (var, f))
def postoji(var, f): return Formula(TipFormula.EGZISTENCIJALNI, (var, f))

# Primjeri
print("Strukture logike predikata:")
print("============================")
print("\nTermovi:")
print(f"  konstante: {konst('sokrat')}, {konst('atena')}, {konst('5')}")
print(f"  varijable: {var('x')}, {var('y')}, {var('z')}")
print(f"  funkcije: {funk('otac', konst('sokrat'))}, {funk('plus', konst('2'), konst('3'))}")

print("\nAtomarne formule:")
print(f"  {pred('Covjek', konst('sokrat'))} - 'Sokrat je čovjek'")
print(f"  {pred('Voli', var('x'), var('y'))} - 'x voli y'")
print(f"  {jednako(var('x'), var('y'))} - 'x je jednak y'")

print("\nKvantificirane formule:")
print(f"  {svaki('x', pred('Smrtan', var('x')))} - 'Svi su smrtni'")
```

```
print(f"  {postoji('x', pred('Mudrac', var('x')))} - 'Postoji mudrac'")
```

Output

```
Strukture logike predikata:
============================

Termovi:
  konstante: sokrat, atena, 5
  varijable: x, y, z
  funkcije: otac(sokrat), plus(2, 3)

Atomarne formule:
  Covjek(sokrat) - 'Sokrat je čovjek'
  Voli(x, y) - 'x voli y'
  x = y - 'x je jednak y'

Kvantificirane formule:
  ∀x Smrtan(x) - 'Svi su smrtni'
  ∃x Mudrac(x) - 'Postoji mudrac'
```

4.1.2 Tarskijeva semantička teorija istine

Tarski je formulirao svoju čuvenu **T-konvenciju** koja definira uvjete adekvatnosti za definiciju istine:

> "'Snijeg je bijel' je istinito ako i samo ako je snijeg bijel" - Tarski, 1933

Ova naizgled trivijalna tvrdnja skriva duboku istinu: istina se definira kroz **metajezik** koji govori o **objektnom jeziku**.

Za logiku predikata, trebamo definirati:

- **Domenu** (univerzum diskursa)
- **Interpretaciju** konstanti, funkcija i predikata
- **Valuaciju** varijabli

Implementirajmo Tarskijevu semantiku:

```
@dataclass
class Model:
    """Tarskijeva struktura za interpretaciju logike predikata."""
    domena: Set[Any]
```

```
    interpretacija: Dict[str, Any]

    def interpretiraj_konstantu(self, simbol: str) -> Any:
        """Interpretira konstantu."""
        if simbol in self.interpretacija:
            return self.interpretacija[simbol]
        raise ValueError(f"Konstanta {simbol} nije definirana")

    def interpretiraj_funkciju(self, simbol: str, argumenti: List[Any]) -> Any:
        """Interpretira funkcijski simbol."""
        if simbol in self.interpretacija:
            funkcija = self.interpretacija[simbol]
            if callable(funkcija):
                return funkcija(*argumenti)
            # Za jednostavne slučajeve, možemo koristiti rječnik
            if isinstance(funkcija, dict):
                kljuc = tuple(argumenti) if len(argumenti) > 1 else argumenti[0]
                return funkcija.get(kljuc)
        raise ValueError(f"Funkcija {simbol} nije definirana")

    def interpretiraj_predikat(self, simbol: str, argumenti: List[Any]) -> bool:
        """Provjerava je li predikat istinit za dane argumente."""
        if simbol in self.interpretacija:
            ekstenzija = self.interpretacija[simbol]
            if len(argumenti) == 0:
                return ekstenzija  # Propozicijska konstanta
            elif len(argumenti) == 1:
                return argumenti[0] in ekstenzija
            else:
                return tuple(argumenti) in ekstenzija
        return False

@dataclass
class Valuacija:
    """Pridjeljuje vrijednosti varijablama."""
    vrijednosti: Dict[str, Any]

    def __getitem__(self, varijabla: str) -> Any:
        return self.vrijednosti.get(varijabla)

    def __setitem__(self, varijabla: str, vrijednost: Any):
        self.vrijednosti[varijabla] = vrijednost

    def kopija(self) -> 'Valuacija':
        """Stvara kopiju valuacije."""
        return Valuacija(self.vrijednosti.copy())

    def promijeni(self, varijabla: str, vrijednost: Any) -> 'Valuacija':
        """Vraća novu valuaciju s promijenjenom varijablom."""
        nova = self.kopija()
        nova[varijabla] = vrijednost
        return nova

```

```python
def evaluiraj_term(term: Term, model: Model, valuacija: Valuacija) -> Any:
    """Evaluira term u modelu s danom valuacijom."""
    if term.tip == TipTerm.KONSTANTA:
        return model.interpretiraj_konstantu(term.simbol)
    elif term.tip == TipTerm.VARIJABLA:
        return valuacija[term.simbol]
    elif term.tip == TipTerm.FUNKCIJA:
        arg_vrijednosti = [evaluiraj_term(arg, model, valuacija)
                           for arg in term.argumenti]
        return model.interpretiraj_funkciju(term.simbol, arg_vrijednosti)
    raise ValueError(f"Nepoznat tip terma: {term.tip}")

# Primjer Tarskijeve strukture
filozofi_model = Model(
    domena={'sokrat', 'platon', 'aristotel'},
    interpretacija={
        'sokrat': 'sokrat',
        'platon': 'platon',
        'aristotel': 'aristotel',
        'Filozof': {'sokrat', 'platon', 'aristotel'},
        'Ucitelj': {('sokrat', 'platon'), ('platon', 'aristotel')},
        'Smrtan': {'sokrat', 'platon', 'aristotel'},
        'ucitelj': {'sokrat': 'platon', 'platon': 'aristotel'}
    }
)

val = Valuacija({'x': 'sokrat', 'y': 'platon'})

print("Tarskijeva struktura (model):")
print("==============================")
print(f"\nDomena D = {filozofi_model.domena}")
print("\nInterpretacija I:")
for simbol in ['sokrat', 'platon', 'Filozof', 'Ucitelj', 'Smrtan']:
    if simbol in filozofi_model.interpretacija:
        print(f"  I({simbol}) = {filozofi_model.interpretacija[simbol]}")

print("\nValuacija v:")
for var1, val_var in val.vrijednosti.items():
    print(f"  v({var1}) = {val_var}")

print("\nEvaluacija termova:")
t1 = konst('sokrat')
t2 = var('x')
t3 = funk('ucitelj', var('x'))

print(f"  [[{t1}]]^{{M,v}} = {evaluiraj_term(t1, filozofi_model, val)}")
print(f"  [[{t2}]]^{{M,v}} = {evaluiraj_term(t2, filozofi_model, val)}")
print(f"  [[{t3}]]^{{M,v}} = {evaluiraj_term(t3, filozofi_model, val)}")
```

Output

```
Tarskijeva struktura (model):
==============================

Domena D = {'platon', 'aristotel', 'sokrat'}

Interpretacija I:
  I(sokrat) = sokrat
  I(platon) = platon
  I(Filozof) = {'platon', 'aristotel', 'sokrat'}
  I(Ucitelj) = {('platon', 'aristotel'), ('sokrat', 'platon')}
  I(Smrtan) = {'platon', 'aristotel', 'sokrat'}

Valuacija v:
  v(x) = sokrat
  v(y) = platon

Evaluacija termova:
  [[sokrat]]^{M,v} = sokrat
  [[x]]^{M,v} = sokrat
  [[ucitelj(x)]]^{M,v} = platon
```

4.1.3 Semantička evaluacija formula

Tarskijeva rekurzivna definicija istine definira kada je formula φ istinita u modelu $\mathcal{M}$ s valuacijom v, što *pi*šemo $\mathcal{M}, v \models \varphi$:

1. $\mathcal{M}, v \models P(t_1, ..., t_n)$ ako $(I(t_1), ..., I(t_n)) \in I(P)$
2. $\mathcal{M}, v \models \neg\varphi$ ako $\mathcal{M}, v \not\models \varphi$
3. $\mathcal{M}, v \models \varphi \wedge \psi$ ako $\mathcal{M}, v \models \varphi$ i $\mathcal{M}, v \models \psi$
4. $\mathcal{M}, v \models \forall x\varphi$ ako za svaki $d \in D$: $\mathcal{M}, v[x/d] \models \varphi$
5. $\mathcal{M}, v \models \exists x\varphi$ ako postoji $d \in D$: $\mathcal{M}, v[x/d] \models \varphi$

Implementirajmo ovu rekurzivnu definiciju:

```python
def evaluiraj_formulu(formula: Formula, model: Model, valuacija: Valuacija) -> bool:
    """Rekurzivno evaluira formulu prema Tarskijevoj definiciji."""

    if formula.tip == TipFormula.PREDIKAT:
        simbol, termovi = formula.sadrzaj
        arg_vrijednosti = [evaluiraj_term(t, model, valuacija) for t in termovi]
        return model.interpretiraj_predikat(simbol, arg_vrijednosti)

```

```
    elif formula.tip == TipFormula.JEDNAKOST:
        t1, t2 = formula.sadrzaj
        v1 = evaluiraj_term(t1, model, valuacija)
        v2 = evaluiraj_term(t2, model, valuacija)
        return v1 == v2

    elif formula.tip == TipFormula.NEGACIJA:
        return not evaluiraj_formulu(formula.sadrzaj, model, valuacija)

    elif formula.tip == TipFormula.KONJUNKCIJA:
        f1, f2 = formula.sadrzaj
        return (evaluiraj_formulu(f1, model, valuacija) and
                evaluiraj_formulu(f2, model, valuacija))

    elif formula.tip == TipFormula.DISJUNKCIJA:
        f1, f2 = formula.sadrzaj
        return (evaluiraj_formulu(f1, model, valuacija) or
                evaluiraj_formulu(f2, model, valuacija))

    elif formula.tip == TipFormula.IMPLIKACIJA:
        f1, f2 = formula.sadrzaj
        return (not evaluiraj_formulu(f1, model, valuacija) or
                evaluiraj_formulu(f2, model, valuacija))

    elif formula.tip == TipFormula.UNIVERZALNI:
        varijabla, podformula = formula.sadrzaj
        # Provjeri za sve elemente domene
        for element in model.domena:
            nova_valuacija = valuacija.promijeni(varijabla, element)
            if not evaluiraj_formulu(podformula, model, nova_valuacija):
                return False
        return True

    elif formula.tip == TipFormula.EGZISTENCIJALNI:
        varijabla, podformula = formula.sadrzaj
        # Provjeri postoji li barem jedan element
        for element in model.domena:
            nova_valuacija = valuacija.promijeni(varijabla, element)
            if evaluiraj_formulu(podformula, model, nova_valuacija):
                return True
        return False

    raise ValueError(f"Nepoznat tip formule: {formula.tip}")

def prikazi_evaluaciju(formula: Formula, model: Model, valuacija: Valuacija):
    """Prikazuje rezultat evaluacije."""
    rezultat = evaluiraj_formulu(formula, model, valuacija)
    simbol = "⊤" if rezultat else "⊥"
    print(f"  M, v ⊨ {formula} = {simbol}")

# Testiranje evaluacije
print("Evaluacija formula:")
print("====================")
```

```
62
63 print("\nAtomarne formule:")
64 f1 = pred('Filozof', konst('sokrat'))
65 f2 = pred('Ucitelj', konst('sokrat'), konst('platon'))
66 f3 = pred('Ucitelj', konst('platon'), konst('sokrat'))
67
68 prikazi_evaluaciju(f1, filozofi_model, val)
69 prikazi_evaluaciju(f2, filozofi_model, val)
70 prikazi_evaluaciju(f3, filozofi_model, val)
71
72 print("\nSložene formule:")
73 f4 = konj(pred('Filozof', var('x')), pred('Smrtan', var('x')))
74 f5 = impl(pred('Filozof', var('x')), pred('Smrtan', var('x')))
75
76 prikazi_evaluaciju(f4, filozofi_model, val)
77 prikazi_evaluaciju(f5, filozofi_model, val)
78
79 print("\nKvantificirane formule:")
80 f6 = svaki('x', pred('Smrtan', var('x')))
81 f7 = postoji('x', pred('Ucitelj', var('x'), konst('platon')))
82 f8 = svaki('x', impl(pred('Filozof', var('x')), pred('Smrtan', var('x'))))
83 f9 = postoji('x', postoji('y', pred('Ucitelj', var('x'), var('y'))))
84 f10 = svaki('x', postoji('y', pred('Ucitelj', var('x'), var('y'))))
85
86 prikazi_evaluaciju(f6, filozofi_model, val)
87 prikazi_evaluaciju(f7, filozofi_model, val)
88 prikazi_evaluaciju(f8, filozofi_model, val)
89 prikazi_evaluaciju(f9, filozofi_model, val)
90 prikazi_evaluaciju(f10, filozofi_model, val)
```

Output

```
Evaluacija formula:
====================

Atomarne formule:
  M, v ⊨ Filozof(sokrat) = ⊤
  M, v ⊨ Ucitelj(sokrat, platon) = ⊤
  M, v ⊨ Ucitelj(platon, sokrat) = ⊥

Složene formule:
  M, v ⊨ (Filozof(x) ∧ Smrtan(x)) = ⊤
  M, v ⊨ (Filozof(x) → Smrtan(x)) = ⊤

Kvantificirane formule:
  M, v ⊨ ∀x Smrtan(x) = ⊤
  M, v ⊨ ∃x Ucitelj(x, platon) = ⊤
  M, v ⊨ ∀x (Filozof(x) → Smrtan(x)) = ⊤
  M, v ⊨ ∃x ∃y Ucitelj(x, y) = ⊤
  M, v ⊨ ∀x ∃y Ucitelj(x, y) = ⊥
```

4.1.4 Slobodne i vezane varijable

Quine je naglasio važnost razlikovanja **slobodnih** i **vezanih** varijabli:

> "Biti znači biti vrijednost vezane varijable" (*To be is to be the value of a bound variable*) - Quine, 1948

Varijabla je **vezana** ako je u dosegu kvantifikatora, inače je **slobodna**.

Formula bez slobodnih varijabli naziva se **zatvorena formula** ili **rečenica**.

```python
def slobodne_varijable(formula: Formula, vezane: Set[str] = None) -> Set[str]:
    """Vraća skup slobodnih varijabli u formuli."""
    if vezane is None:
        vezane = set()

    if formula.tip == TipFormula.PREDIKAT:
        _, termovi = formula.sadrzaj
        slobodne = set()
        for term in termovi:
            if term.tip == TipTerm.VARIJABLA and term.simbol not in vezane:
                slobodne.add(term.simbol)
            elif term.tip == TipTerm.FUNKCIJA:
                slobodne.update(term.slobodne_varijable() - vezane)
        return slobodne

    elif formula.tip == TipFormula.JEDNAKOST:
        t1, t2 = formula.sadrzaj
        slobodne = set()
        if t1.tip == TipTerm.VARIJABLA and t1.simbol not in vezane:
            slobodne.add(t1.simbol)
        if t2.tip == TipTerm.VARIJABLA and t2.simbol not in vezane:
            slobodne.add(t2.simbol)
        return slobodne

    elif formula.tip == TipFormula.NEGACIJA:
        return slobodne_varijable(formula.sadrzaj, vezane)

    elif formula.tip in [TipFormula.KONJUNKCIJA, TipFormula.DISJUNKCIJA,
    ↪  TipFormula.IMPLIKACIJA]:
        f1, f2 = formula.sadrzaj
        return slobodne_varijable(f1, vezane) | slobodne_varijable(f2, vezane)

    elif formula.tip in [TipFormula.UNIVERZALNI, TipFormula.EGZISTENCIJALNI]:
        varijabla, podformula = formula.sadrzaj
        nove_vezane = vezane | {varijabla}
        return slobodne_varijable(podformula, nove_vezane)

    return set()
```

```
38
39 def vezane_varijable(formula: Formula) -> Set[str]:
40     """Vraća skup vezanih varijabli u formuli."""
41     vezane = set()
42
43     if formula.tip == TipFormula.NEGACIJA:
44         return vezane_varijable(formula.sadrzaj)
45
46     elif formula.tip in [TipFormula.KONJUNKCIJA, TipFormula.DISJUNKCIJA,
    ↪  TipFormula.IMPLIKACIJA]:
47         f1, f2 = formula.sadrzaj
48         return vezane_varijable(f1) | vezane_varijable(f2)
49
50     elif formula.tip in [TipFormula.UNIVERZALNI, TipFormula.EGZISTENCIJALNI]:
51         varijabla, podformula = formula.sadrzaj
52         return {varijabla} | vezane_varijable(podformula)
53
54     return vezane
55
56 def je_zatvorena(formula: Formula) -> bool:
57     """Provjerava je li formula zatvorena (rečenica)."""
58     return len(slobodne_varijable(formula)) == 0
59
60 def substituiraj(formula: Formula, varijabla: str, term: Term) -> Formula:
61     """Substituira term za varijablu u formuli."""
62     if formula.tip == TipFormula.PREDIKAT:
63         simbol, termovi = formula.sadrzaj
64         novi_termovi = []
65         for t in termovi:
66             if t.tip == TipTerm.VARIJABLA and t.simbol == varijabla:
67                 novi_termovi.append(term)
68             else:
69                 novi_termovi.append(t)
70         return pred(simbol, *novi_termovi)
71
72     elif formula.tip == TipFormula.NEGACIJA:
73         return neg(substituiraj(formula.sadrzaj, varijabla, term))
74
75     elif formula.tip in [TipFormula.KONJUNKCIJA, TipFormula.DISJUNKCIJA,
    ↪  TipFormula.IMPLIKACIJA]:
76         f1, f2 = formula.sadrzaj
77         nova_f1 = substituiraj(f1, varijabla, term)
78         nova_f2 = substituiraj(f2, varijabla, term)
79         if formula.tip == TipFormula.KONJUNKCIJA:
80             return konj(nova_f1, nova_f2)
81         elif formula.tip == TipFormula.DISJUNKCIJA:
82             return disj(nova_f1, nova_f2)
83         else:
84             return impl(nova_f1, nova_f2)
85
86     elif formula.tip in [TipFormula.UNIVERZALNI, TipFormula.EGZISTENCIJALNI]:
87         kvant_var, podformula = formula.sadrzaj
88         if kvant_var == varijabla:
```

```
            # Varijabla je vezana, ne substituiraj
            return formula
        nova_podformula = substituiraj(podformula, varijabla, term)
        if formula.tip == TipFormula.UNIVERZALNI:
            return svaki(kvant_var, nova_podformula)
        else:
            return postoji(kvant_var, nova_podformula)

    return formula

# Testiranje
print("Analiza varijabli:")
print("==================")

formule_test = [
    pred('Voli', var('x'), var('y')),
    svaki('x', pred('Voli', var('x'), var('y'))),
    svaki('x', postoji('y', pred('Voli', var('x'), var('y')))),
    impl(svaki('x', pred('P', var('x'))), pred('P', var('y')))
]

for f in formule_test:
    slobodne = slobodne_varijable(f)
    vezane = vezane_varijable(f)
    zatvorena = "⊤" if je_zatvorena(f) else "⊥"
    print(f"\nFormula: {f}")
    print(f"  Slobodne varijable: {slobodne}")
    print(f"  Vezane varijable: {vezane}")
    print(f"  Zatvorena? {zatvorena}")

print("\nSubstitucija:")
print("=============")

f1 = pred('P', var('x'))
f2 = svaki('x', pred('P', var('x')))
f3 = svaki('x', pred('P', var('x'), var('y')))

print(f"\n{f1}[x/sokrat] = {substituiraj(f1, 'x', konst('sokrat'))}")
print(f"{f2}[x/sokrat] = {substituiraj(f2, 'x', konst('sokrat'))}")
print(f"{f3}[y/sokrat] = {substituiraj(f3, 'y', konst('sokrat'))}")
```

Output

```
Analiza varijabli:
==================

Formula: Voli(x, y)
  Slobodne varijable: {'y', 'x'}
  Vezane varijable: set()
  Zatvorena? ⊥
```

```
Formula: ∀x Voli(x, y)
  Slobodne varijable: {'y'}
  Vezane varijable: {'x'}
  Zatvorena? ⊥

Formula: ∀x ∃y Voli(x, y)
  Slobodne varijable: set()
  Vezane varijable: {'y', 'x'}
  Zatvorena? ⊤

Formula: (∀x P(x) → P(y))
  Slobodne varijable: {'y'}
  Vezane varijable: {'x'}
  Zatvorena? ⊥

Substitucija:
=============

P(x)[x/sokrat] = P(sokrat)
∀x P(x)[x/sokrat] = ∀x P(x)
∀x P(x, y)[y/sokrat] = ∀x P(x, sokrat)
```

4.1.5 Valjanost i zadovoljivost

U logici predikata razlikujemo:

- **Valjanost** (logički istinita) formula: istinita u svim modelima
- **Zadovoljiva** formula: istinita u barem jednom modelu
- **Nezadovoljiva** formula: lažna u svim modelima

GÃ¶del je dokazao potpunost logike predikata prvog reda:

> "Svaka valjana formula logike predikata prvog reda je dokaziva" - Goedel, 1929

Ali također i nepotpunost aritmetike:

> "U svakom dovoljno bogatom formalnom sustavu postoje istinite tvrdnje koje se ne mogu dokazati" - GÃ¶del, 1931

```python
def je_validna(formula: Formula, modeli: List[Model]) -> bool:
    """Provjerava je li formula validna u svim danim modelima."""
    for model in modeli:
```

```
        # Za svaku moguću valuaciju
        # (za jednostavnost, provjeravamo samo praznu valuaciju za zatvorene formule)
        if je_zatvorena(formula):
            val = Valuacija({})
            if not evaluiraj_formulu(formula, model, val):
                return False
        else:
            # Za formule sa slobodnim varijablama, trebamo provjeriti sve valuacije
            slobodne = slobodne_varijable(formula)

            def sve_valuacije(varijable, domena, trenutna={}):
                if not varijable:
                    yield Valuacija(trenutna.copy())
                else:
                    var = varijable[0]
                    for element in domena:
                        trenutna[var] = element
                        yield from sve_valuacije(varijable[1:], domena, trenutna)
                        del trenutna[var]

            for val in sve_valuacije(list(slobodne), model.domena):
                if not evaluiraj_formulu(formula, model, val):
                    return False
    return True

def je_zadovoljiva(formula: Formula, modeli: List[Model]) -> bool:
    """Provjerava je li formula zadovoljiva u barem jednom modelu."""
    for model in modeli:
        if je_zatvorena(formula):
            val = Valuacija({})
            if evaluiraj_formulu(formula, model, val):
                return True
        else:
            slobodne = slobodne_varijable(formula)

            def sve_valuacije(varijable, domena, trenutna={}):
                if not varijable:
                    yield Valuacija(trenutna.copy())
                else:
                    var = varijable[0]
                    for element in domena:
                        trenutna[var] = element
                        yield from sve_valuacije(varijable[1:], domena, trenutna)
                        del trenutna[var]

            for val in sve_valuacije(list(slobodne), model.domena):
                if evaluiraj_formulu(formula, model, val):
                    return True
    return False

# Testni modeli
model1 = Model(
    domena={'a', 'b'},
```

```
    interpretacija={
        'P': {'a'},
        'R': {('a', 'b'), ('b', 'a')}
    }
)

model2 = Model(
    domena={'1', '2'},
    interpretacija={
        'P': set(),
        'R': {('1', '1')}
    }
)

model3 = Model(
    domena={'x', 'y', 'z'},
    interpretacija={
        'P': {'x', 'y'},
        'R': {('x', 'y'), ('y', 'y'), ('z', 'y')}
    }
)

modeli = [model1, model2, model3]

# Test formula
print("Provjera validnosti i zadovoljivosti:")
print("====================================")

test_formule = [
    ("Zakon identiteta", svaki('x', impl(pred('P', var('x')), pred('P', var('x'))))),
    ("Egzistencija P", postoji('x', pred('P', var('x')))),
    ("Kontradikcija", konj(svaki('x', pred('P', var('x'))),
                           neg(svaki('x', pred('P', var('x')))))),
    ("Svaki ima nekoga", svaki('x', postoji('y', pred('R', var('x'), var('y'))))),
    ("Postoji za sve", postoji('y', svaki('x', pred('R', var('x'), var('y')))))
]

for naziv, formula in test_formule:
    print(f"\nFormula: {formula}")

    # Evaluiraj u svakom modelu
    rezultati = []
    for i, model in enumerate(modeli, 1):
        val = Valuacija({})
        rez = evaluiraj_formulu(formula, model, val)
        rezultati.append(rez)
        print(f"  Model {i}: {'⊤' if rez else '⊥'}")

    # Odredi status
    if all(rezultati):
        print("  Status: VALIDNA ✓")
    elif any(rezultati):
        print("  Status: ZADOVOLJIVA")
```

```
    else:
        print("  Status: NEZADOVOLJIVA ×")

print("\nVažno zapažanje:")
print("  ∀x∃y R(x,y) ≠ ∃y∀x R(x,y)")
print("  Redoslijed kvantifikatora je bitan!")
```

Output

```
Provjera validnosti i zadovoljivosti:
=====================================

Formula: ∀x (P(x) → P(x))
  Model 1: ⊤
  Model 2: ⊤
  Model 3: ⊤
  Status: VALIDNA ✓

Formula: ∃x P(x)
  Model 1: ⊤
  Model 2: ⊥
  Model 3: ⊤
  Status: ZADOVOLJIVA

Formula: (∀x P(x) ∧ ¬∀x P(x))
  Model 1: ⊥
  Model 2: ⊥
  Model 3: ⊥
  Status: NEZADOVOLJIVA ×

Formula: ∀x ∃y R(x, y)
  Model 1: ⊤
  Model 2: ⊥
  Model 3: ⊤
  Status: ZADOVOLJIVA

Formula: ∃y ∀x R(x, y)
  Model 1: ⊥
  Model 2: ⊥
  Model 3: ⊤
  Status: ZADOVOLJIVA

Važno zapažanje:
  ∀x∃y R(x,y) ≠ ∃y∀x R(x,y)
  Redoslijed kvantifikatora je bitan!
```

4.1.6 Skolemizacija i prenex normalna forma

Thoralf Skolem pokazao je kako eliminirati egzistencijalne kvantifikatore:

> "Svaka formula logike predikata može se transformirati u ekvivalentnu formulu bez egzistencijalnih kvantifikatora" - Skolem, 1920

Prenex normalna forma: svi kvantifikatori su na početku formule.

Skolemizacija: zamjena egzistencijalnih kvantifikatora Skolemovim funkcijama.

```python
def u_prenex_formu(formula: Formula, kvantifikatori=[]) -> Formula:
    """Pretvara formulu u prenex normalnu formu.
    (Pojednostavljena verzija za demonstraciju)
    """
    # Ovo je pojednostavljena implementacija
    # Potpuna implementacija bi trebala rukovati svim slučajevima

    if formula.tip in [TipFormula.UNIVERZALNI, TipFormula.EGZISTENCIJALNI]:
        var, podformula = formula.sadrzaj
        return u_prenex_formu(podformula, kvantifikatori + [(formula.tip, var)])

    # Rekonstruiraj formulu s kvantifikatorima na početku
    rezultat = formula
    for tip_kvant, var in reversed(kvantifikatori):
        if tip_kvant == TipFormula.UNIVERZALNI:
            rezultat = svaki(var, rezultat)
        else:
            rezultat = postoji(var, rezultat)

    return rezultat

def skolemizuj(formula: Formula, univerzalni_kontekst=[]) -> Formula:
    """Skolemizacija - eliminacija egzistencijalnih kvantifikatora.
    (Pojednostavljena verzija)
    """

    if formula.tip == TipFormula.UNIVERZALNI:
        var, podformula = formula.sadrzaj
        nova_podformula = skolemizuj(podformula, univerzalni_kontekst + [var])
        return svaki(var, nova_podformula)

    elif formula.tip == TipFormula.EGZISTENCIJALNI:
        var, podformula = formula.sadrzaj

        # Stvori Skolemov term
        if not univerzalni_kontekst:
            # Skolemova konstanta
            skolem_term = konst(f"c_{var}")
        else:
            # Skolemova funkcija
            argumenti = [var(v) for v in univerzalni_kontekst]
            skolem_term = funk(f"f_{var}", *argumenti)

```

```
        # Substituiraj
        nova_podformula = substituiraj(podformula, var, skolem_term)
        return skolemizuj(nova_podformula, univerzalni_kontekst)

    elif formula.tip == TipFormula.NEGACIJA:
        return neg(skolemizuj(formula.sadrzaj, univerzalni_kontekst))

    elif formula.tip in [TipFormula.KONJUNKCIJA, TipFormula.DISJUNKCIJA,
    ↪ TipFormula.IMPLIKACIJA]:
        f1, f2 = formula.sadrzaj
        nova_f1 = skolemizuj(f1, univerzalni_kontekst)
        nova_f2 = skolemizuj(f2, univerzalni_kontekst)
        if formula.tip == TipFormula.KONJUNKCIJA:
            return konj(nova_f1, nova_f2)
        elif formula.tip == TipFormula.DISJUNKCIJA:
            return disj(nova_f1, nova_f2)
        else:
            return impl(nova_f1, nova_f2)

    return formula

print("Prenex normalna forma:")
print("======================")

# Primjeri prenex transformacije
f1 = impl(svaki('x', pred('P', var('x'))), postoji('y', pred('Q', var('y'))))
print(f"\nOriginal: {f1}")
print(f"Prenex: ∃x ∃y (¬P(x) ∨ Q(y))")

f2 = svaki('x', impl(pred('P', var('x')), postoji('y', pred('R', var('x'), var('y')))))
print(f"\nOriginal: {f2}")
print(f"Prenex: ∀x ∃y (¬P(x) ∨ R(x, y))")

print("\nSkolemizacija:")
print("===============")

# Primjer 1: Egzistencijalni kvantifikator bez univerzalnog konteksta
f3 = postoji('x', pred('P', var('x')))
print(f"\nOriginal: {f3}")
print(f"Skolemizovano: P(c)")
print("  gdje je c nova Skolemova konstanta")

# Primjer 2: Egzistencijalni u kontekstu univerzalnog
f4 = svaki('x', postoji('y', pred('R', var('x'), var('y'))))
print(f"\nOriginal: {f4}")
print(f"Skolemizovano: ∀x R(x, f(x))")
print("  gdje je f nova Skolemova funkcija")

# Primjer 3: Složeniji slučaj
f5 = postoji('x', svaki('y', postoji('z', pred('S', var('x'), var('y'), var('z')))))
print(f"\nOriginal: {f5}")
print(f"Skolemizovano: ∀y S(a, y, g(y))")
print("  gdje su a konstanta i g funkcija")
```

```
96
97 print("\nKorak po korak - formula: ∀x (P(x) → ∃y R(x, y))")
98 print("================================================")
99 print("\n1. Eliminacija implikacije:")
100 print("   ∀x (¬P(x) ∨ ∃y R(x, y))")
101 print("\n2. Premještanje kvantifikatora (prenex):")
102 print("   ∀x ∃y (¬P(x) ∨ R(x, y))")
103 print("\n3. Skolemizacija:")
104 print("   ∀x (¬P(x) ∨ R(x, f(x)))")
105 print("   gdje f(x) je Skolemova funkcija")
106 print("\n4. Rezultat je ekvizadovoljiv s originalnom formulom")
```

Output

```
Prenex normalna forma:
======================

Original: (∀x P(x) → ∃y Q(y))
Prenex: ∃x ∃y (¬P(x) ∨ Q(y))

Original: ∀x (P(x) → ∃y R(x, y))
Prenex: ∀x ∃y (¬P(x) ∨ R(x, y))

Skolemizacija:
==============

Original: ∃x P(x)
Skolemizovano: P(c)
  gdje je c nova Skolemova konstanta

Original: ∀x ∃y R(x, y)
Skolemizovano: ∀x R(x, f(x))
  gdje je f nova Skolemova funkcija

Original: ∃x ∀y ∃z S(x, y, z)
Skolemizovano: ∀y S(a, y, g(y))
  gdje su a konstanta i g funkcija

Korak po korak - formula: ∀x (P(x) → ∃y R(x, y))
================================================

1. Eliminacija implikacije:
   ∀x (¬P(x) ∨ ∃y R(x, y))

2. Premještanje kvantifikatora (prenex):
   ∀x ∃y (¬P(x) ∨ R(x, y))

3. Skolemizacija:
   ∀x (¬P(x) ∨ R(x, f(x)))
   gdje f(x) je Skolemova funkcija
```

```
4. Rezultat je ekvizadovoljiv s originalnom formulom
```

4.1.7 Herbrandov univerzum i teorem

Jacques Herbrand pokazao je kako svesti problem zadovoljivosti na propozicijski slučaj:

> "Zadovoljivost formule prvog reda može se svesti na zadovoljivost beskonačnog skupa propozicijskih formula" - Herbrand, 1930

Herbrandov univerzum: skup svih termova koji se mogu konstruirati iz konstanti i funkcija.

```python
def generiraj_herbrandov_univerzum(konstante: Set[str], funkcije: Dict[str, int], dubina: int = 2):
    """Generira Herbrandov univerzum do zadane dubine.
    funkcije: dict koji mapira ime funkcije u njen aritet
    """
    H = []

    # H_0 - samo konstante
    H.append(set(konstante))

    for d in range(1, dubina + 1):
        novi = set(H[d-1])  # Uključi sve iz prethodne razine

        # Za svaku funkciju
        for funk_ime, aritet in funkcije.items():
            # Generiraj sve kombinacije argumenata iz H[d-1]
            import itertools
            for args in itertools.product(H[d-1], repeat=aritet):
                args_str = ", ".join(args)
                novi.add(f"{funk_ime}({args_str})")

        H.append(novi)

    return H

def generiraj_herbrandovu_bazu(predikati: Dict[str, int], termovi: Set[str]):
    """Generira Herbrandovu bazu - sve atomarne formule s Herbrandovim termovima.
    predikati: dict koji mapira ime predikata u njegov aritet
    """
    baza = set()

    for pred_ime, aritet in predikati.items():
        if aritet == 0:
            baza.add(pred_ime)
        else:
```

```
            import itertools
            for args in itertools.product(termovi, repeat=aritet):
                args_str = ", ".join(args)
                baza.add(f"{pred_ime}({args_str})")

    return baza

# Primjer
konstante = {'a', 'b'}
funkcije = {'f': 1, 'g': 2}  # f je unarna, g je binarna
predikati = {'P': 1, 'R': 2}  # P je unarni, R je binarni

print("Herbrandov univerzum:")
print("=====================")
print(f"\nKonstante: {konstante}")
print(f"Funkcije: {{{', '.join(f'{f}/{a}' for f, a in funkcije.items())}}}")

H = generiraj_herbrandov_univerzum(konstante, funkcije, 2)

print(f"\nH₀ = {H[0]}")
print(f"H₁ = {H[1]}")
print(f"H₂ = ... ({len(H[2])} termova)")

print("\nHerbrandova baza:")
print("==================")

baza = generiraj_herbrandovu_bazu(predikati, H[0])
print(f"\nZa predikate P/1 i R/2:")
print("B₀ = {")
baza_lista = sorted(list(baza))
print(f"  {', '.join(baza_lista[:2])},")
print(f"  {', '.join(baza_lista[2:])}")
print("}")

print("\nHerbrandove interpretacije:")
print("============================")
print(f"\nBroj mogućih interpretacija za B₀: 2^6 = {2**len(baza)}")

print("\nPrimjer interpretacije I₁:")
print("  P(a) = ⊤, P(b) = ⊥")
print("  R(a, a) = ⊤, R(a, b) = ⊤")
print("  R(b, a) = ⊥, R(b, b) = ⊤")

print("\nHerbrandov teorem:")
print("==================")
print("\nFormula ∀x ∃y R(x, y) je zadovoljiva")
print("⟺")
print("Skup {R(a, f(a)), R(b, f(b)), ...} je zadovoljiv")
print("\nTime se problem prvog reda svodi na propozicijski!")
```

4.1.8 Praktična primjena: Mini dokazivač teorema

Implementirajmo jednostavan dokazivač teorema koji koristi rezoluciju:

```python
class Klauzula:
    """Predstavlja klauzulu u CNF formi."""
    def __init__(self, literali):
        self.literali = set(literali)

    def __repr__(self):
        if not self.literali:
            return "□"  # Prazna klauzula
        return "{" + ", ".join(str(l) for l in self.literali) + "}"

    def je_prazna(self):
        return len(self.literali) == 0

class Literal:
    """Predstavlja literal (atomarna formula ili njena negacija)."""
    def __init__(self, predikat, argumenti, negiran=False):
        self.predikat = predikat
        self.argumenti = argumenti
        self.negiran = negiran

    def __repr__(self):
        args = "(" + ", ".join(str(a) for a in self.argumenti) + ")" if self.argumenti else
        ↪  ""
        return ("¬" if self.negiran else "") + self.predikat + args

    def __eq__(self, other):
        return (self.predikat == other.predikat and
                self.argumenti == other.argumenti and
                self.negiran == other.negiran)

    def __hash__(self):
        return hash((self.predikat, tuple(self.argumenti), self.negiran))

    def komplementaran(self, other):
        """Provjerava jesu li literali komplementarni."""
        return (self.predikat == other.predikat and
                self.argumenti == other.argumenti and
                self.negiran != other.negiran)

def unifikacija(t1, t2, subst={}):
    """Jednostavna unifikacija za konstante i varijable."""
    if t1 == t2:
        return subst
    elif t1.startswith('x') or t1.startswith('y') or t1.startswith('z'):
        # t1 je varijabla
        if t1 in subst:
            return unifikacija(subst[t1], t2, subst)
```

```
        else:
            subst[t1] = t2
            return subst
    elif t2.startswith('x') or t2.startswith('y') or t2.startswith('z'):
        # t2 je varijabla
        return unifikacija(t2, t1, subst)
    else:
        return None  # Unifikacija neuspješna

def primijeni_substituciju(literal, subst):
    """Primjenjuje substituciju na literal."""
    novi_argumenti = []
    for arg in literal.argumenti:
        if arg in subst:
            novi_argumenti.append(subst[arg])
        else:
            novi_argumenti.append(arg)
    return Literal(literal.predikat, novi_argumenti, literal.negiran)

# Demonstracija
print("Rezolucijski dokazivač teorema:")
print("================================")
print("\nCilj: Dokazati da Sokrat je smrtan")
print("\nPremise:")
print("  1. ∀x (Covjek(x) → Smrtan(x))")
print("  2. Covjek(sokrat)")

# Pretvorba u klauzule
k1 = Klauzula([Literal("Covjek", ['x'], True), Literal("Smrtan", ['x'])])
k2 = Klauzula([Literal("Covjek", ['sokrat'])])
k3 = Klauzula([Literal("Smrtan", ['sokrat'], True)])  # Negacija cilja

print("\nKlauzule (nakon Skolemizacije i CNF):")
print(f"  C1: {k1}")
print(f"  C2: {k2}")
print(f"  C3: {k3} (negacija cilja)")

print("\nRezolucijski dokaz:")
print("-------------------")

print("Korak 1: Rezolucija C1 i C2")
print("  Unifikacija: x ↦ sokrat")
print("  {¬Covjek(sokrat), Smrtan(sokrat)} + {Covjek(sokrat)}")
print("  ⟹ {Smrtan(sokrat)}")

print("\nKorak 2: Rezolucija {Smrtan(sokrat)} i C3")
print("  {Smrtan(sokrat)} + {¬Smrtan(sokrat)}")
print("  ⟹ □ (prazna klauzula)")

print("\n✓ DOKAZ PRONAĐEN!")
print("Sokrat je doista smrtan.")

print("\n========================================")
```

```
print("\nSloženiji primjer - tranzitivnost:")
print("==================================")
print("\nPremise:")
print("  1. ∀x ∀y (Roditelj(x, y) → Predak(x, y))")
print("  2. ∀x ∀y ∀z ((Predak(x, y) ∧ Predak(y, z)) → Predak(x, z))")
print("  3. Roditelj(ivan, marko)")
print("  4. Roditelj(marko, ana)")
print("\nCilj: Dokazati Predak(ivan, ana)")
print("\nKoraci dokazivanja:")
print("1. Iz premise 1 i 3: Predak(ivan, marko)")
print("2. Iz premise 1 i 4: Predak(marko, ana)")
print("3. Iz premise 2, koraka 1 i 2: Predak(ivan, ana)")
print("\n✓ Teorem dokazan!")
```

4.1.9 Zadaci za vježbu

Za produbljivanje razumijevanja semantike i sintakse logike predikata, predlažemo sljedeće vježbe:

Implementacija potpune evaluacije

Proširite funkciju "evaluiraj formulu" da podržava složenije termovima s ugniježđenim funkcijama.

Provjera validnosti formula

Implementirajte algoritam koji provjerava je li formula validna konstruiranjem konačno mnogo modela (za formule s konačnim Herbrandovim univerzumom).

Unifikacijski algoritam

Implementirajte potpuni Robinson unifikacijski algoritam koji rukuje složenim termovima i provjerava occur-check.

CNF transformacija

Napišite funkciju koja pretvara proizvolju formulu logike predikata u konjunktivnu normalnu formu (CNF).

Rezolucijski dokazivač

Proširite mini dokazivač da automatski pronalazi dokaze korištenjem rezolucije s unifikacijom.

Model checker

Stvorite interaktivni model checker gdje korisnik može definirati model i provjeriti istinitost formula.

Analiza složenosti

Istražite složenost problema zadovoljivosti za različite fragmente logike predikata (Horn klauzule, monadski predikat, itd.).

Visualizacija modela

Implementirajte grafički prikaz modela kao usmjerenog grafa za binarne relacije.

Prevođenje prirodnog jezika

Napišite parser koji prevodi jednostavne rečenice prirodnog jezika u formule logike predikata.

Igra evaluacije

Implementirajte Hintikkinu semantičku igru za evaluaciju formula - dvoje igrača (Svatko i Netko) naizmjence biraju vrijednosti za kvantificirane varijable.

Svaki zadatak postupno gradi razumijevanje Tarskijeve semantičke teorije istine i odnosa između sintakse i semantike u logici predikata prvog reda.

4.1.10 Zaključak

Kroz ovu implementaciju istražili smo Tarskijev revolucionarni pristup semantici logike predikata:

1. **Sintaksu logike predikata** - termove, formule i kvantifikatore
2. **Tarskijevu semantičku teoriju istine** - modele, interpretacije i valuacije
3. **Rekurzivnu evaluaciju** formula prema Tarskijevoj definiciji
4. **Validnost i zadovoljivost** u logici predikata

5. **Skolemizaciju** i vezu s propozicijskim slučajem

Tarski je svojim radom odgovorio na fundamentalno pitanje:

> "Što znači da je rečenica istinita?" - Tarski, 1933

Njegov odgovor kroz rekurzivnu definiciju istine omogućio je:

- Precizno razumijevanje semantike formalnih jezika
- Razvoj teorije modela
- Temelj za automatsko dokazivanje teorema
- Most između logike i računarstva

Tarskijev svijet pokazuje kako apstraktni matematički objekti mogu biti reprezentirani i manipulirani kroz konkretne strukture podataka. Python implementacija omogućava nam da eksperimentiramo s ovim konceptima i razvijemo intuiciju za duboke logičke istine.

Logika predikata prvog reda ostaje temelj:

- **Baza podataka** - SQL je u biti logika predikata
- **Umjetne inteligencije** - predstavljanje znanja
- **Verifikacije programa** - dokazivanje svojstava
- **Semantičkog weba** - formalizacija ontologija

Tarskijev svijet nas uči da istina nije samo filozofski koncept, već precizno definirana matematička struktura koju možemo konstruirati, analizirati i računati.

Poglavlje 5

Turingov svijet: Granice izračunljivosti

5.1 Od Hilbertovog programa do Turingovih strojeva

Godine 1928., David Hilbert postavio je svoj čuveni **Entscheidungsproblem** (problem odlučivosti):

> "Postoji li algoritam koji za svaku tvrdnju logike predikata može odlučiti je li dokaziva?" - Hilbert, 1928

Alan Turing (1912-1954) odgovorio je na ovo pitanje 1936. godine uvođenjem koncepta koji danas nazivamo **Turingov stroj**:

> "Moguće je izmisliti jedan stroj koji se može koristiti za računanje bilo kojeg izračunljivog niza" - Turing, *On Computable Numbers*, 1936

Turingov rad ne samo da je riješio Hilbertov problem (negativno!), već je postavio temelje moderne teorije računanja.

5.1.1 Turingov stroj - formalni model računanja

Turingov stroj sastoji se od:

- **Beskonačne trake** podijeljene na ćelije
- **Glave za čitanje/pisanje** koja se može pomicati lijevo ili desno
- **Konačnog skupa stanja** uključujući početno i završna stanja
- **Prijelazne funkcije** koja određuje sljedeći korak

Formalno: $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$

Implementirajmo Turingov stroj u Pythonu:

```python
from dataclasses import dataclass
from typing import Dict, Tuple, Optional, List, Set
from enum import Enum

class Smjer(Enum):
    """Smjer pomicanja glave."""
    LIJEVO = 'L'
    DESNO = 'D'
    STOJ = 'S'

@dataclass
class TuringovStroj:
    """Implementacija determinističkog Turingovog stroja."""

    def __init__(self, stanja: Set[str], ulazni_alfabet: Set[str],
                 alfabet_trake: Set[str], prijelazi: Dict,
                 pocetno: str, prihvaca: str, odbacuje: str):
        self.Q = stanja
        self.Sigma = ulazni_alfabet
        self.Gamma = alfabet_trake
        self.delta = prijelazi
        self.q0 = pocetno
        self.q_accept = prihvaca
        self.q_reject = odbacuje
        self.prazno = '_'  # Simbol za praznu ćeliju

        # Trenutna konfiguracija
        self.traka = []
        self.pozicija = 0
        self.stanje = self.q0
        self.povijest = []

    def postavi_ulaz(self, ulaz: str):
        """Postavlja ulazni niz na traku."""
        self.traka = list(ulaz) + [self.prazno]
        self.pozicija = 0
        self.stanje = self.q0
        self.povijest = [self._trenutna_konfiguracija()]

    def _trenutna_konfiguracija(self) -> Tuple[str, List[str], int]:
        """Vraća trenutnu konfiguraciju (stanje, traka, pozicija)."""
        return (self.stanje, self.traka.copy(), self.pozicija)

    def korak(self) -> bool:
        """Izvršava jedan korak stroja. Vraća False ako je završio."""
        if self.stanje in [self.q_accept, self.q_reject]:
            return False
```

```
48
49          # Čitaj simbol s trake
50          if self.pozicija >= len(self.traka):
51              self.traka.append(self.prazno)
52          simbol = self.traka[self.pozicija]
53
54          # Pronađi prijelaz
55          if (self.stanje, simbol) not in self.delta:
56              self.stanje = self.q_reject
57              return False
58
59          novo_stanje, novi_simbol, smjer = self.delta[(self.stanje, simbol)]
60
61          # Ažuriraj konfiguraciju
62          self.stanje = novo_stanje
63          self.traka[self.pozicija] = novi_simbol
64
65          if smjer == Smjer.LIJEVO:
66              self.pozicija = max(0, self.pozicija - 1)
67          elif smjer == Smjer.DESNO:
68              self.pozicija += 1
69              if self.pozicija >= len(self.traka):
70                  self.traka.append(self.prazno)
71
72          self.povijest.append(self._trenutna_konfiguracija())
73          return True
74
75      def pokreni(self, max_koraka: int = 1000) -> str:
76          """Pokreće stroj do kraja ili maksimalnog broja koraka."""
77          koraci = 0
78          while self.korak() and koraci < max_koraka:
79              koraci += 1
80
81          if self.stanje == self.q_accept:
82              return "PRIHVAĆEN"
83          elif self.stanje == self.q_reject:
84              return "ODBAČEN"
85          else:
86              return "PREKORAČEN LIMIT"
87
88      def prikazi_povijest(self):
89          """Prikazuje povijest izvršavanja."""
90          for i, (stanje, traka, poz) in enumerate(self.povijest):
91              # Prikaži traku s trenutnom pozicijom
92              prikaz = ""
93              for j, simbol in enumerate(traka):
94                  if j == poz:
95                      prikaz += f"[{stanje}] {simbol} "
96                  else:
97                      prikaz += f"{simbol} "
98              print(f"Korak {i}: {prikaz}")
99              if poz < len(traka):
100                 print(" " * (9 + poz * 2 + len(stanje) // 2) + "↑")
```

```

# Primjer: Stroj koji mijenja 0→1 i 1→0
stanja = {'q0', 'q1', 'q_accept', 'q_reject'}
ulazni = {'0', '1'}
traka = {'0', '1', '_'}

# Prijelazna funkcija
prijelazi = {
    ('q0', '0'): ('q1', '1', Smjer.DESNO),
    ('q0', '1'): ('q1', '0', Smjer.DESNO),
    ('q1', '0'): ('q0', '1', Smjer.DESNO),
    ('q1', '1'): ('q0', '0', Smjer.DESNO),
    ('q0', '_'): ('q_accept', '_', Smjer.LIJEVO),
    ('q1', '_'): ('q_accept', '_', Smjer.LIJEVO)
}

tm = TuringovStroj(stanja, ulazni, traka, prijelazi, 'q0', 'q_accept', 'q_reject')

print("Turingov stroj inicijaliziran")
print("=====================================")
print("\nKomponente stroja:")
print(f"  Stanja Q = {tm.Q}")
print(f"  Ulazni alfabet Σ = {tm.Sigma}")
print(f"  Alfabet trake Γ = {tm.Gamma}")
print(f"  Početno stanje = {tm.q0}")
print(f"  Prihvaća stanje = {tm.q_accept}")
print(f"  Odbacuje stanje = {tm.q_reject}")

print("\nPrijelazna funkcija δ:")
for (s, sym), (ns, nsym, d) in list(prijelazi.items())[:3]:
    print(f"  δ({s}, {sym}) = ({ns}, {nsym}, {d.value})")

# Test
ulaz = "0011"
print(f"\nSimulacija na ulazu '{ulaz}':")
print("==============================")
print()

tm.postavi_ulaz(ulaz)
rezultat = tm.pokreni()
tm.prikazi_povijest()

print(f"\nRezultat: {rezultat}")
print(f"Finalna traka: {''.join(tm.traka)}")
```

Output

```
Turingov stroj inicijaliziran
=====================================

Komponente stroja:
```

```
  Stanja Q = {'q0', 'q1', 'q_reject', 'q_accept'}
  Ulazni alfabet Σ = {'1', '0'}
  Alfabet trake Γ = {'1', '0', '_'}
  Početno stanje = q0
  Prihvaća stanje = q_accept
  Odbacuje stanje = q_reject

Prijelazna funkcija δ:
  δ(q0, 0) = (q1, 1, D)
  δ(q0, 1) = (q1, 0, D)
  δ(q1, 0) = (q0, 1, D)

Simulacija na ulazu '0011':
============================

Korak 0: [q0] 0 0 1 1 _
          ↑
Korak 1: 1 [q1] 0 1 1 _
             ↑
Korak 2: 1 1 [q0] 1 1 _
               ↑
Korak 3: 1 1 0 [q1] 1 _
                 ↑
Korak 4: 1 1 0 0 [q0] _
                   ↑
Korak 5: 1 1 0 [q_accept] 0 _
                     ↑

Rezultat: PRIHVAĆEN
Finalna traka: 1100_
```

5.1.2 Church-Turingova teza

Istovremeno s Turingom, Alonzo Church razvio je lambda račun kao alternativni formalizam:

> "Efektivno izračunljiva funkcija je ona koja se može izraziti u lambda računu" - Church, 1936

Church-Turingova teza tvrdi da su svi razumni modeli računanja ekvivalentni:

- Turingovi strojevi
- Lambda račun
- Rekurzivne funkcije (Gödel, Kleene)
- Postovi sustavi

- Register strojevi

Implementirajmo lambda račun i pokažimo ekvivalenciju:

```python
class LambdaTerm:
    """Apstraktna klasa za lambda terme."""
    pass

class Var(LambdaTerm):
    """Varijabla."""
    def __init__(self, ime):
        self.ime = ime

    def __repr__(self):
        return self.ime

    def substitute(self, var, term):
        if self.ime == var:
            return term
        return self

class Abs(LambdaTerm):
    """Lambda apstrakcija."""
    def __init__(self, param, tijelo):
        self.param = param
        self.tijelo = tijelo

    def __repr__(self):
        return f"λ{self.param}.{self.tijelo}"

    def substitute(self, var, term):
        if self.param == var:
            return self  # Varijabla je vezana
        return Abs(self.param, self.tijelo.substitute(var, term))

class App(LambdaTerm):
    """Aplikacija."""
    def __init__(self, funkcija, argument):
        self.funkcija = funkcija
        self.argument = argument

    def __repr__(self):
        return f"({self.funkcija} {self.argument})"

    def substitute(self, var, term):
        return App(
            self.funkcija.substitute(var, term),
            self.argument.substitute(var, term)
        )

def beta_redukcija(term: LambdaTerm, max_koraka=100) -> LambdaTerm:
```

```
    """Beta redukcija lambda terma."""
    koraci = 0

    while koraci < max_koraka:
        if isinstance(term, App) and isinstance(term.funkcija, Abs):
            # Beta redukcija: (λx.M)N → M[x/N]
            term = term.funkcija.tijelo.substitute(
                term.funkcija.param,
                term.argument
            )
            koraci += 1
        else:
            break

    return term

# Church brojevi - reprezentacija prirodnih brojeva
def church_broj(n):
    """Konstruira Church broj za n."""
    # n = λf.λx.f^n(x)
    f = lambda func: lambda x: x if n == 0 else func(church_broj(n-1)(func)(x))
    return f

def church_to_int(church_n):
    """Pretvara Church broj u Python int."""
    return church_n(lambda x: x + 1)(0)

# Aritmetičke operacije
SUCC = lambda n: lambda f: lambda x: f(n(f)(x))
PLUS = lambda m: lambda n: lambda f: lambda x: m(f)(n(f)(x))
MULT = lambda m: lambda n: lambda f: m(n(f))

# Booleove vrijednosti
TRUE = lambda x: lambda y: x
FALSE = lambda x: lambda y: y
NOT = lambda p: p(FALSE)(TRUE)
AND = lambda p: lambda q: p(q)(FALSE)
OR = lambda p: lambda q: p(TRUE)(q)

print("Lambda račun")
print("============")
print("\nOsnovni konstrukti:")
print(f"  Varijabla: {Var('x')}")
print(f"  Apstrakcija: {Abs('x', Var('x'))}")
print(f"  Aplikacija: {App(Var('f'), Var('x'))}")

print("\nChurch brojevi:")
for i in range(4):
    if i == 0:
        print(f"  {i} = λf.λx.x")
    else:
        f_aplikacije = ' '.join(['(f' for _ in range(i)]) + ' x' + ')' * i
        print(f"  {i} = λf.λx.{f_aplikacije}")
```

```
101
102 print("\nAritmetičke operacije:")
103 print("  SUCC = λn.λf.λx.(f ((n f) x))")
104 print("  PLUS = λm.λn.λf.λx.((m f) ((n f) x))")
105 print("  MULT = λm.λn.λf.(m (n f))")
106
107 print("\nPrimjer redukcije: (SUCC 1)")
108 print("================================")
109 print("\nKorak 0: ((λn.λf.λx.(f ((n f) x)) λf.λx.(f x))")
110 print("Korak 1: λf.λx.(f ((λf.λx.(f x) f) x))")
111 print("Korak 2: λf.λx.(f (λx.(f x) x))")
112 print("Korak 3: λf.λx.(f (f x))")
113 print("\nRezultat: Church broj 2")
114
115 # Python simulacija
116 print("\nPython simulacija Church brojeva:")
117 print("================================")
118
119 for i in range(4):
120     c = church_broj(i)
121     print(f"church({i}) kao Python broj: {church_to_int(c)}")
122
123 # Testovi
124 print("\nTestovi:")
125 c2 = church_broj(2)
126 c3 = church_broj(3)
127
128 rezultat_succ = church_to_int(SUCC(c2))
129 print(f"  succ(2) = {rezultat_succ} {'✓' if rezultat_succ == 3 else '×'}")
130
131 rezultat_plus = church_to_int(PLUS(c2)(c3))
132 print(f"  plus(2, 3) = {rezultat_plus} {'✓' if rezultat_plus == 5 else '×'}")
133
134 rezultat_mult = church_to_int(MULT(c2)(c3))
135 print(f"  mult(2, 3) = {rezultat_mult} {'✓' if rezultat_mult == 6 else '×'}")
136
137 print("\nBooleove vrijednosti:")
138 print("  TRUE = λx.λy.x")
139 print("  FALSE = λx.λy.y")
140 print("  NOT = λp.((p FALSE) TRUE)")
141 print("  AND = λp.λq.((p q) FALSE)")
142
143 # Test Booleovih operacija
144 def bool_to_python(church_bool):
145     return church_bool(True)(False)
146
147 print(f"\nTest: NOT TRUE = FALSE {'✓' if not bool_to_python(NOT(TRUE)) else '×'}")
148 print(f"Test: AND TRUE FALSE = FALSE {'✓' if not bool_to_python(AND(TRUE)(FALSE)) else
↪  '×'}")
```

Output

```
Lambda račun
============

Osnovni konstrukti:
  Varijabla: x
  Apstrakcija: λx.x
  Aplikacija: (f x)

Church brojevi:
  0 = λf.λx.x
  1 = λf.λx.(f x)
  2 = λf.λx.(f (f x))
  3 = λf.λx.(f (f (f x)))

Aritmetičke operacije:
  SUCC = λn.λf.λx.(f ((n f) x))
  PLUS = λm.λn.λf.λx.((m f) ((n f) x))
  MULT = λm.λn.λf.(m (n f))

Primjer redukcije: (SUCC 1)
================================

Korak 0: ((λn.λf.λx.(f ((n f) x)) λf.λx.(f x))
Korak 1: λf.λx.(f ((λf.λx.(f x) f) x))
Korak 2: λf.λx.(f (λx.(f x) x))
Korak 3: λf.λx.(f (f x))

Rezultat: Church broj 2

Python simulacija Church brojeva:
================================
church(0) kao Python broj: 0
church(1) kao Python broj: 1
church(2) kao Python broj: 2
church(3) kao Python broj: 3

Testovi:
  succ(2) = 3 ✓
  plus(2, 3) = 5 ✓
  mult(2, 3) = 6 ✓

Booleove vrijednosti:
  TRUE = λx.λy.x
  FALSE = λx.λy.y
  NOT = λp.((p FALSE) TRUE)
  AND = λp.λq.((p q) FALSE)

Test: NOT TRUE = FALSE ✓
Test: AND TRUE FALSE = FALSE ✓
```

5.1.3 Problem zaustavljanja - prva granica izračunljivosti

Turing je 1936. dokazao da ne postoji algoritam koji može odlučiti hoće li se proizvoljan program zaustaviti:

> "Ne može postojati opći proces za određivanje hoće li se dani stroj ikada ispisati 0" - Turing, 1936

Halting problem: Za dani Turingov stroj M i ulaz w, odlučiti hoće li se M zaustaviti na w.

Dokaz koristi **dijagonalizaciju**, tehniku koju je Cantor koristio za dokaz neprebrojavosti realnih brojeva:

```python
def simuliraj_halt_problem():
    """Demonstracija paradoksa problema zaustavljanja."""

    # Pretpostavljamo da imamo magičnu funkciju HALT
    def pretpostavljeni_halt(program_kod, ulaz):
        """Hipotetička funkcija koja 'rješava' problem zaustavljanja."""
        # Ovo je nemoguće implementirati općenito!
        # Za demonstraciju, vraćamo random vrijednost
        import hashlib
        h = hashlib.md5((program_kod + ulaz).encode()).hexdigest()
        return int(h, 16) % 2 == 0

    # Dijagonalizacijski stroj D
    def D(M_kod):
        """Stroj koji stvara paradoks."""
        if pretpostavljeni_halt(M_kod, M_kod):
            # Ako M staje na M, onda D ulazi u beskonačnu petlju
            while True:
                pass
        else:
            # Ako M ne staje na M, onda D staje
            return "STOP"

    # Paradoks: Što se događa s D(D)?
    D_kod = "def D(M): ..."

    return D_kod

print("Problem zaustavljanja")
print("=====================")
print("\nPretpostavimo da postoji funkcija HALT(M, w) koja vraća:")
print("  ⊤ ako se M zaustavlja na ulazu w")
print("  ⊥ ako se M ne zaustavlja na ulazu w")

print("\nDijagonalizacijski dokaz:")
print("-------------------------")
```

```
print("\n1. Konstruiramo stroj D koji na ulazu M:")
print("   - Ako HALT(M, M) = ⊤, onda D ulazi u beskonačnu petlju")
print("   - Ako HALT(M, M) = ⊥, onda D se zaustavlja")

print("\n2. Što se događa kad pokrenemo D(D)?")
print("   - Ako D(D) se zaustavlja, onda HALT(D, D) = ⊤")
print("     → Po definiciji D, to znači da D(D) ne staje!")
print("   - Ako D(D) ne staje, onda HALT(D, D) = ⊥")
print("     → Po definiciji D, to znači da D(D) staje!")

print("\n3. KONTRADIKCIJA! ⋆")
print("\nDakle, funkcija HALT ne može postojati.")

# Simulacija
print("\nSimulacija paradoksa:")
print("====================")

def simple_loop():
    while True:
        pass

def simple_halt():
    return "STOP"

print("\nPokušaj 1: Pretpostavljamo HALT(simple_loop, '') = ⊥")
print("  Stroj D bi se zaustavio")

print("\nPokušaj 2: Pretpostavljamo HALT(simple_halt, '') = ⊤")
print("  Stroj D bi ušao u petlju")

print("\nPokušaj 3: HALT(D, D) = ?")
print("  Paradoks! Ne možemo odlučiti.")

print("\nPraktične posljedice:")
print("====================")
print("\nNeodlučivi problemi u programiranju:")
print("  • Hoće li program ikad pristupiti null pointeru?")
print("  • Jesu li dva programa ekvivalentna?")
print("  • Hoće li program ikad ispisati \"Hello World\"?")
print("  • Je li funkcija totalna (definirana za sve ulaze)?")
print("  • Postoji li ulaz za koji program vraća 42?")
print("\nRice-ov teorem: Svako netrivijalno svojstvo programa je neodlučivo!")
```

Output

```
Problem zaustavljanja
=====================

Pretpostavimo da postoji funkcija HALT(M, w) koja vraća:
  ⊤ ako se M zaustavlja na ulazu w
  ⊥ ako se M ne zaustavlja na ulazu w
```

```
Dijagonalizacijski dokaz:
-------------------------

1. Konstruiramo stroj D koji na ulazu M:
   - Ako HALT(M, M) = ⊤, onda D ulazi u beskonačnu petlju
   - Ako HALT(M, M) = ⊥, onda D se zaustavlja

2. Što se događa kad pokrenemo D(D)?
   - Ako D(D) se zaustavlja, onda HALT(D, D) = ⊤
     → Po definiciji D, to znači da D(D) ne staje!
   - Ako D(D) ne staje, onda HALT(D, D) = ⊥
     → Po definiciji D, to znači da D(D) staje!

3. KONTRADIKCIJA! ⋆

Dakle, funkcija HALT ne može postojati.

Simulacija paradoksa:
=====================

Pokušaj 1: Pretpostavljamo HALT(simple_loop, '') = ⊥
  Stroj D bi se zaustavio

Pokušaj 2: Pretpostavljamo HALT(simple_halt, '') = ⊤
  Stroj D bi ušao u petlju

Pokušaj 3: HALT(D, D) = ?
  Paradoks! Ne možemo odlučiti.

Praktične posljedice:
=====================

Neodlučivi problemi u programiranju:
  • Hoće li program ikad pristupiti null pointeru?
  • Jesu li dva programa ekvivalentna?
  • Hoće li program ikad ispisati "Hello World"?
  • Je li funkcija totalna (definirana za sve ulaze)?
  • Postoji li ulaz za koji program vraća 42?

Rice-ov teorem: Svako netrivijalno svojstvo programa je neodlučivo!
```

5.1.4 Rekurzivno prebrojive vs. odlučive jezike

Turing je razlikovao dvije klase problema:

- **Odlučivi** (rekurzivni): Turingov stroj uvijek staje s DA/NE odgovorom
- **Poluodlučivi** (rekurzivno prebrojivi): Turingov stroj staje za DA, možda ne staje za

NE

Post je pokazao:

> "Jezik je odlučiv ako i samo ako su i on i njegov komplement rekurzivno prebrojivi"
> - Post, 1944

```python
class JezikKlasa(Enum):
    """Klasifikacija jezika po odlučivosti."""
    ODLUCIV = "odlučiv"
    POLUODLUCIV = "poluodlučiv"
    NEODLUCIV = "neodlučiv"

def odluciv_paran_broj_jedinica(w: str) -> str:
    """Odlučuje jezik s parnim brojem jedinica."""
    broj_jedinica = w.count('1')
    if broj_jedinica % 2 == 0:
        return "PRIHVAĆEN"
    else:
        return "ODBAČEN"

def poluodluciv_halting(M_opis: str, w: str, max_koraka: int = 100) -> str:
    """Simulira poluodlučivost problema zaustavljanja."""
    # Simuliramo izvršavanje do max_koraka
    # U stvarnosti, ovo bi moglo trajati zauvijek!

    koraci = 0
    while koraci < max_koraka:
        # Simulacija...
        koraci += 1

        # Za demonstraciju, "zaustavlja se" za neke slučajeve
        if len(M_opis + w) % 7 == 0:
            return "PRIHVAĆEN"

    return "NEPOZNATO (još uvijek radi...)"

def enumerator(jezik_generator):
    """Enumerator za rekurzivno prebrojiv jezik."""
    for niz in jezik_generator():
        yield niz

def generator_anbn():
    """Generira jezik {a^n b^n takav da n ≥ 0}."""
    n = 0
    while True:
        yield 'a' * n + 'b' * n
        n += 1

def provjeri_pripadnost_enumeracijom(w: str, enumerator, max_koraka: int = 1000):
```

```
    """Provjerava pripadnost enumeracijom."""
    for i, generirani in enumerate(enumerator):
        if i >= max_koraka:
            return "NEPOZNATO"
        if generirani == w:
            return "PRONAĐEN"
    return "NIJE PRONAĐEN"

print("Hijerarhija jezika")
print("==================")
print("\n1. ODLUČIVI (Rekurzivni) jezici:")
print("   Turingov stroj M takav da za svaki w:")
print("   • M(w) = PRIHVATI ako w ∈ L")
print("   • M(w) = ODBACI ako w ∉ L")
print("   • M se UVIJEK zaustavlja")

print("\n2. POLUODLUČIVI (Rekurzivno prebrojivi) jezici:")
print("   Turingov stroj M takav da za svaki w:")
print("   • M(w) = PRIHVATI ako w ∈ L")
print("   • M(w) = ODBACI ili ∞ ako w ∉ L")

print("\n3. NEODLUČIVI jezici:")
print("   Ne postoji Turingov stroj koji odlučuje jezik")

print("\nPrimjeri:")
print("=========")

print("\nODLUČIV: L = {w takav da w ima paran broj jedinica}")
testovi = ['1011', '1001', '1111']
for w in testovi:
    rezultat = odluciv_paran_broj_jedinica(w)
    broj = w.count('1')
    print(f"  Test '{w}': {broj} jedinice → {rezultat}")

print("\nPOLUODLUČIV: H = {⟨M,w⟩ takav da M se zaustavlja na w}")
print("  Možemo simulirati M na w")
print("  Ako stane → PRIHVATI")
print("  Ako ne stane → ... (čekamo zauvijek)")

print("\nSimulacija poluodlučivog problema:")
print("==================================")

print("\nEnumerator za jezik {a^n b^n takav da n ≥ 0}:")
gen = generator_anbn()
prvih_6 = [next(gen) for _ in range(6)]
print(f"  Generirani nizovi: {', '.join(prvih_6 if prvih_6[0] else ['ε'] + prvih_6[1:])}")

print("\nProvjera 'aabb' ∈ L?")
enum = enumerator(generator_anbn)
for i in range(3):
    niz = next(enum)
    if niz == 'aabb':
        print(f"  Korak {i+1}: Generiraj {niz if niz else 'ε'} → PRONAĐEN! ✓")
```

```
        break
    else:
        print(f"  Korak {i+1}: Generiraj {niz if niz else 'ε'} → ne odgovara")

print("\nProvjera 'abab' ∈ L?")
enum = enumerator(generator_anbn)
for i in range(5):
    niz = next(enum)
    print(f"  Korak {i+1}: {niz if niz else 'ε'} → ne")
print("  ... (nikad neće pronaći, ali ne znamo to unaprijed!)")

print("\nPostov teorem:")
print("==============")
print("\nL je odlučiv ⟺ L i ne L su rekurzivno prebrojivi")
print("\nDokaz (⟹):")
print("  Ako je L odlučiv, imamo M koji uvijek staje.")
print("  Za L: pokreni M, prihvati ako M prihvati.")
print("  Za ne L: pokreni M, prihvati ako M odbaci.")
print("\nDokaz (⟸):")
print("  Ako su L i ne L r.p., imamo M₁ i M₂.")
print("  Simuliraj M₁ i M₂ paralelno (dovetailing).")
print("  Jedan mora stati → odluči!")
```

Output

```
Hijerarhija jezika
==================

1. ODLUČIVI (Rekurzivni) jezici:
   Turingov stroj M takav da za svaki w:
   • M(w) = PRIHVATI ako w ∈ L
   • M(w) = ODBACI ako w ∉ L
   • M se UVIJEK zaustavlja

2. POLUODLUČIVI (Rekurzivno prebrojivi) jezici:
   Turingov stroj M takav da za svaki w:
   • M(w) = PRIHVATI ako w ∈ L
   • M(w) = ODBACI ili ∞ ako w ∉ L

3. NEODLUČIVI jezici:
   Ne postoji Turingov stroj koji odlučuje jezik

Primjeri:
=========

ODLUČIV: L = {w takav da w ima paran broj jedinica}
  Test '1011': 3 jedinice → ODBAČEN
  Test '1001': 2 jedinice → PRIHVAĆEN
  Test '1111': 4 jedinice → PRIHVAĆEN

POLUODLUČIV: H = {⟨M,w⟩ takav da M se zaustavlja na w}
```

```
  Možemo simulirati M na w
  Ako stane → PRIHVATI
  Ako ne stane → ... (čekamo zauvijek)

Simulacija poluodlučivog problema:
==================================

Enumerator za jezik {a^n b^n takav da n ≥ 0}:
  Generirani nizovi: ε, ab, aabb, aaabbb, aaaabbbb, aaaaabbbbb

Provjera 'aabb' ∈ L?
  Korak 1: Generiraj ε → ne odgovara
  Korak 2: Generiraj ab → ne odgovara
  Korak 3: Generiraj aabb → PRONAĐEN! ✓

Provjera 'abab' ∈ L?
  Korak 1: ε → ne
  Korak 2: ab → ne
  Korak 3: aabb → ne
  Korak 4: aaabbb → ne
  Korak 5: aaaabbbb → ne
  ... (nikad neće pronaći, ali ne znamo to unaprijed!)

Postov teorem:
==============

L je odlučiv ⟺ L i ne L su rekurzivno prebrojivi

Dokaz (⟹):
  Ako je L odlučiv, imamo M koji uvijek staje.
  Za L: pokreni M, prihvati ako M prihvati.
  Za ne L: pokreni M, prihvati ako M odbaci.

Dokaz (⟸):
  Ako su L i ne L r.p., imamo M_1 i M_2.
  Simuliraj M_1 i M_2 paralelno (dovetailing).
  Jedan mora stati → odluči!
```

5.1.5 Redukcije i stupnjevi neodlučivosti

Turing je uveo koncept **redukcije** - svođenja jednog problema na drugi:

> "Mnogi problemi mogu se svesti na problem zaustavljanja" - Turing, 1936

Many-one redukcija: $A \leq_m B$ ako postoji izračunljiva funkcija f takva da:

$$w \in A \iff f(w) \in B$$

```python
class Redukcija:
    """Reprezentira many-one redukciju između problema."""

    def __init__(self, ime, od_problema, do_problema):
        self.ime = ime
        self.od = od_problema
        self.do = do_problema

    def __repr__(self):
        return f"{self.od} $\leq_m$ {self.do}"

def reduciraj_acceptance_na_halting(M, w):
    """Reducira ACCEPTANCE problem na HALTING."""

    # Konstruiramo novi stroj M'
    def M_prime(x):
        # Simuliraj M na w
        rezultat = simuliraj_tm(M, w)

        if rezultat == "PRIHVAĆEN":
            return "STOP"  # M' staje
        else:
            while True:  # M' ne staje
                pass

    return M_prime

def simuliraj_tm(M, w, max_koraka=100):
    """Simulira Turingov stroj (pojednostavljeno)."""
    # Za demonstraciju
    import hashlib
    h = int(hashlib.md5((str(M) + w).encode()).hexdigest(), 16)
    if h % 3 == 0:
        return "PRIHVAĆEN"
    elif h % 3 == 1:
        return "ODBAČEN"
    else:
        return "LOOP"

class OracleTM:
    """Turingov stroj s orakulom."""

    def __init__(self, oracle_problem):
        self.oracle = oracle_problem

    def query_oracle(self, instance):
        """Pita orakul za odgovor."""
        # U teoriji, orakul instantno odgovara
        return self.oracle(instance)

    def riješi_problem_s_orakulom(self, problem, ulaz):
        """Rješava problem koristeći orakul."""
```

```
        # Primjer: s H-orakulom možemo riješiti mnoge probleme
        if problem == "EMPTY":
            # Je li L(M) prazan?
            # Konstruiraj M' koji prihvaća sve ako M prihvaća bar nešto
            return self.query_oracle(("modified_M", ulaz))
        return "NEPOZNAT"

def halting_oracle(instance):
    """Hipotetički halting oracle."""
    M, w = instance
    # Ovo je nemoguće implementirati!
    # Za demonstraciju:
    if "while True" in str(M):
        return False
    elif "return" in str(M):
        return True
    else:
        return None

print("Redukcije među problemima")
print("=========================")
print("\nMany-one redukcija: A ≤ₘ B")
print("  Postoji izračunljiva f: w ∈ A ⟺ f(w) ∈ B")
print("\nAko A ≤ₘ B i B je odlučiv → A je odlučiv")
print("Ako A ≤ₘ B i A je neodlučiv → B je neodlučiv")

print("\nPrimjer redukcije:")
print("==================")
print("\nProblem: ACCEPTANCE ≤ₘ HALTING")
print("\nACCEPTANCE: Prihvaća li M ulaz w?")
print("HALTING: Zaustavlja li se M na w?")
print("\nRedukcija:")
print("  Konstruiraj M' koji:")
print("  1. Simulira M na w")
print("  2. Ako M prihvati → M' staje")
print("  3. Ako M odbaci → M' ulazi u petlju")
print("\nTada: M prihvaća w ⟺ M' se zaustavlja na w")

print("\nLanac redukcija:")
print("================")
print("\nEMPTY (Je li L(M) = ∅?)")
print("    ↓ ≤ₘ")
print("REGULAR (Je li L(M) regularan?)")
print("    ↓ ≤ₘ")
print("EQUIVALENT (Je li L(M₁) = L(M₂)?)")
print("    ↓ ≤ₘ")
print("HALTING")
print("\nSvi su neodlučivi!")

print("\nTuringovi stupnjevi:")
print("====================")
print("\nStupanj 0: Odlučivi problemi")
print("  Primjer: Je li broj paran?")
```

```python
print("\nStupanj 0': Problem zaustavljanja")
print("  H = {⟨M,w⟩ takav da M staje na w}")
print("\nStupanj 0'': Halting za oracle strojeve")
print("  H' = {⟨M^H,w⟩ takav da M s H-orakulom staje na w}")
print("\nBeskonačna hijerarhija: 0 < 0' < 0'' < ...")

print("\nOracle simulacija:")
print("==================")

oracle_tm = OracleTM(halting_oracle)

print("\nTuringov stroj s H-orakulom može:")
print("  • Riješiti problem zaustavljanja za obične TM")
print("  • Ali ne može riješiti svoj vlastiti problem zaustavljanja!")

print("\nTest oracle stroja:")
test1 = oracle_tm.query_oracle(("while True: pass", ""))
print(f"  Query: Staje li 'while True: pass'? → {'DA' if test1 else 'NE'}")

test2 = oracle_tm.query_oracle(("return 42", ""))
print(f"  Query: Staje li 'return 42'? → {'DA' if test2 else 'NE'}")

print("  Query: Staje li ovaj oracle stroj? → PARADOKS!")
```

Output

```
Redukcije među problemima
==========================

Many-one redukcija: A ≤ₘ B
  Postoji izračunljiva f: w ∈ A ⟺ f(w) ∈ B

Ako A ≤ₘ B i B je odlučiv → A je odlučiv
Ako A ≤ₘ B i A je neodlučiv → B je neodlučiv

Primjer redukcije:
==================

Problem: ACCEPTANCE ≤ₘ HALTING

ACCEPTANCE: Prihvaća li M ulaz w?
HALTING: Zaustavlja li se M na w?

Redukcija:
  Konstruiraj M' koji:
  1. Simulira M na w
  2. Ako M prihvati → M' staje
  3. Ako M odbaci → M' ulazi u petlju

Tada: M prihvaća w ⟺ M' se zaustavlja na w
```

```
Lanac redukcija:
================

EMPTY (Je li L(M) = ∅?)
    ↓ ≤_m
REGULAR (Je li L(M) regularan?)
    ↓ ≤_m
EQUIVALENT (Je li L(M_1) = L(M_2)?)
    ↓ ≤_m
HALTING

Svi su neodlučivi!

Turingovi stupnjevi:
====================

Stupanj 0: Odlučivi problemi
  Primjer: Je li broj paran?

Stupanj 0': Problem zaustavljanja
  H = {⟨M,w⟩ takav da M staje na w}

Stupanj 0'': Halting za oracle strojeve
  H' = {⟨M^H,w⟩ takav da M s H-orakulom staje na w}

Beskonačna hijerarhija: 0 < 0' < 0'' < ...

Oracle simulacija:
==================

Turingov stroj s H-orakulom može:
  • Riješiti problem zaustavljanja za obične TM
  • Ali ne može riješiti svoj vlastiti problem zaustavljanja!

Test oracle stroja:
  Query: Staje li 'while True: pass'? → NE
  Query: Staje li 'return 42'? → DA
  Query: Staje li ovaj oracle stroj? → PARADOKS!
```

5.1.6 Alternativni modeli: Register strojevi i μ-rekurzivne funkcije

Pokazat ćemo ekvivalenciju različitih modela računanja:

Register strojevi (Shepherdson and Sturgis, 1963): Jednostavniji model s registrima koji drže prirodne brojeve.

μ-rekurzivne funkcije (Gödel, Kleene): Funkcije definirane rekurzijom i minimalizacijom.

```python
class RegisterMachine:
    """Simulacija register stroja."""

    def __init__(self, num_registers=10):
        self.registers = [0] * num_registers
        self.pc = 0  # Program counter
        self.program = []
        self.halted = False

    def inc(self, r):
        """Inkrementiraj registar."""
        self.registers[r] += 1
        self.pc += 1

    def dec(self, r):
        """Dekrementiraj registar (bounded at 0)."""
        if self.registers[r] > 0:
            self.registers[r] -= 1
        self.pc += 1

    def jz(self, r, label):
        """Skoči ako je registar nula."""
        if self.registers[r] == 0:
            self.pc = label
        else:
            self.pc += 1

    def jmp(self, label):
        """Bezuvjetni skok."""
        self.pc = label

    def halt(self):
        """Zaustavi izvršavanje."""
        self.halted = True

    def run(self, max_steps=1000):
        """Pokreni program."""
        steps = 0
        while not self.halted and steps < max_steps and self.pc < len(self.program):
            instruction = self.program[self.pc]
            instruction()
            steps += 1
        return self.registers[0]  # Vraća R0 kao rezultat

# mu-rekurzivne funkcije

def zero(x):
    """Nul funkcija."""
    return 0

def succ(x):
    """Sljedbenik."""
```

```
    return x + 1

def proj(i, *args):
    """Projekcija - vraća i-ti argument."""
    return args[i]

def compose(f, *gs):
    """Kompozicija funkcija."""
    def h(*args):
        g_results = [g(*args) for g in gs]
        return f(*g_results)
    return h

def primitive_recursion(f, g):
    """Primitivna rekurzija."""
    def h(x, *args):
        if x == 0:
            return f(*args)
        else:
            return g(x-1, h(x-1, *args), *args)
    return h

def mu_operator(f):
    """μ-operator - minimalizacija."""
    def h(*args):
        y = 0
        while f(*args, y) != 0:
            y += 1
            if y > 1000:  # Zaštita od beskonačne petlje
                return None
        return y
    return h

# Primjeri μ-rekurzivnih funkcija

# Zbrajanje
add = primitive_recursion(
    lambda y: y,  # add(0, y) = y
    lambda x, rec, y: succ(rec)  # add(S(x), y) = S(add(x, y))
)

# Množenje
mult = primitive_recursion(
    lambda y: 0,  # mult(0, y) = 0
    lambda x, rec, y: add(rec, y)  # mult(S(x), y) = add(mult(x, y), y)
)

# Eksponencijacija
exp = primitive_recursion(
    lambda x: 1,  # exp(x, 0) = 1
    lambda y, rec, x: mult(x, rec)  # exp(x, S(y)) = mult(x, exp(x, y))
)

```

```
print("Register stroj")
print("===============")
print("\nInstrukcije:")
print("  INC(R): R := R + 1")
print("  DEC(R): R := max(0, R - 1)")
print("  JZ(R, L): ako je R = 0, skoči na L")
print("  HALT: zaustavi")

print("\nProgram za zbrajanje R1 + R2 → R0:")
print("==================================")

# Program za zbrajanje
rm = RegisterMachine()
rm.registers[1] = 3  # R1 = 3
rm.registers[2] = 2  # R2 = 2

rm.program = [
    lambda: rm.jz(1, 4),     # 0: ako R1=0, idi na 4
    lambda: rm.dec(1),        # 1: R1--
    lambda: rm.inc(0),        # 2: R0++
    lambda: rm.jmp(0),        # 3: idi na 0
    lambda: rm.jz(2, 8),      # 4: ako R2=0, idi na 8
    lambda: rm.dec(2),        # 5: R2--
    lambda: rm.inc(0),        # 6: R0++
    lambda: rm.jmp(4),        # 7: idi na 4
    lambda: rm.halt()         # 8: HALT
]

print("\n0: JZ(R1, 4)")
print("1: DEC(R1)")
print("2: INC(R0)")
print("3: JMP(0)")
print("4: JZ(R2, 8)")
print("5: DEC(R2)")
print("6: INC(R0)")
print("7: JMP(4)")
print("8: HALT")

print("\nIzvršavanje: R1=3, R2=2")
print("-----------------------")

# Prikaz prvih nekoliko koraka
for i in range(5):
    print(f"Korak {i}: PC={rm.pc}, R0={rm.registers[0]}, R1={rm.registers[1]},
    ↪  R2={rm.registers[2]}")
    if not rm.halted and rm.pc < len(rm.program):
        rm.program[rm.pc]()

print("...")

# Resetuj i pokreni do kraja
rm = RegisterMachine()
rm.registers[1] = 3
```

```
rm.registers[2] = 2
rm.program = [
    lambda: rm.jz(1, 4),
    lambda: rm.dec(1),
    lambda: rm.inc(0),
    lambda: rm.jmp(0),
    lambda: rm.jz(2, 8),
    lambda: rm.dec(2),
    lambda: rm.inc(0),
    lambda: rm.jmp(4),
    lambda: rm.halt()
]

rezultat = rm.run()
print(f"Završeno: R0={rezultat} (3 + 2 = 5) {'✓' if rezultat == 5 else '×'}")

print("\nμ-rekurzivne funkcije")
print("=====================")
print("\nOsnovne funkcije:")
print("  Z(x) = 0 (nul funkcija)")
print("  S(x) = x + 1 (sljedbenik)")
print("  P^n_i(x_1,...,x_n) = x_i (projekcija)")

print("\nOperatori:")
print("  Kompozicija: h(x) = f(g_1(x),...,g_k(x))")
print("  Primitivna rekurzija:")
print("    h(0, y) = f(y)")
print("    h(S(x), y) = g(x, h(x, y), y)")
print("  μ-operator: h(x) = μy[f(x, y) = 0]")

print("\nPrimjer - zbrajanje:")
print("  add(0, y) = y")
print("  add(S(x), y) = S(add(x, y))")
print(f"\nTest: add(3, 2) = {add(3, 2)} {'✓' if add(3, 2) == 5 else '×'}")

print("\nPrimjer - množenje:")
print("  mult(0, y) = 0")
print("  mult(S(x), y) = add(mult(x, y), y)")
print(f"\nTest: mult(3, 4) = {mult(3, 4)} {'✓' if mult(3, 4) == 12 else '×'}")

print("\nPrimjer - eksponencijacija:")
print("  exp(x, 0) = 1")
print("  exp(x, S(y)) = mult(x, exp(x, y))")
print(f"\nTest: exp(2, 3) = {exp(3, 2)} {'✓' if exp(3, 2) == 9 else '×'}")

print("\nμ-operator primjer:")
print("===================")

def sqrt_floor(n):
    """Korijen zaokružen naniže koristeći μ-operator."""
    def predicate(n, x):
        return 0 if x*x > n else 1

```

```
    x = 0
    while predicate(n, x) != 0:
        x += 1
    return x - 1

print("\nsqrt_floor(n) = μx[x² > n]")
print("\nsqrt_floor(10):")
for x in range(5):
    print(f"  x={x}: {x}² = {x*x} {'>' if x*x > 10 else '≤'} 10")
    if x*x > 10:
        print(f"  Rezultat: {x-1} ✓")
        break

print("\nEkvivalencija modela:")
print("====================")
print("\nTuringov stroj ≡ Register stroj ≡ μ-rekurzivne funkcije")
print("\nSvi mogu simulirati jedni druge!")
```

Output

```
Register stroj
==============

Instrukcije:
  INC(R): R := R + 1
  DEC(R): R := max(0, R - 1)
  JZ(R, L): ako je R = 0, skoči na L
  HALT: zaustavi

Program za zbrajanje R1 + R2 → R0:
==================================

0: JZ(R1, 4)
1: DEC(R1)
2: INC(R0)
3: JMP(0)
4: JZ(R2, 8)
5: DEC(R2)
6: INC(R0)
7: JMP(4)
8: HALT

Izvršavanje: R1=3, R2=2
-----------------------
Korak 0: PC=0, R0=0, R1=3, R2=2
Korak 1: PC=1, R0=0, R1=3, R2=2
Korak 2: PC=2, R0=0, R1=2, R2=2
Korak 3: PC=3, R0=1, R1=2, R2=2
Korak 4: PC=0, R0=1, R1=2, R2=2
...
Završeno: R0=5 (3 + 2 = 5) ✓
```

```
μ-rekurzivne funkcije
=====================

Osnovne funkcije:
  Z(x) = 0 (nul funkcija)
  S(x) = x + 1 (sljedbenik)
  P^n_i(x_1,...,x_n) = x_i (projekcija)

Operatori:
  Kompozicija: h(x) = f(g_1(x),...,g_k(x))
  Primitivna rekurzija:
    h(0, y) = f(y)
    h(S(x), y) = g(x, h(x, y), y)
  μ-operator: h(x) = μy[f(x, y) = 0]

Primjer - zbrajanje:
  add(0, y) = y
  add(S(x), y) = S(add(x, y))

Test: add(3, 2) = 5 ✓

Primjer - množenje:
  mult(0, y) = 0
  mult(S(x), y) = add(mult(x, y), y)

Test: mult(3, 4) = 12 ✓

Primjer - eksponencijacija:
  exp(x, 0) = 1
  exp(x, S(y)) = mult(x, exp(x, y))

Test: exp(2, 3) = 8 ×

μ-operator primjer:
===================

sqrt_floor(n) = μx[x^2 > n]

sqrt_floor(10):
  x=0: 0^2 = 0 ≤ 10
  x=1: 1^2 = 1 ≤ 10
  x=2: 2^2 = 4 ≤ 10
  x=3: 3^2 = 9 ≤ 10
  x=4: 4^2 = 16 > 10
  Rezultat: 3 ✓

Ekvivalencija modela:
====================

Turingov stroj ≡ Register stroj ≡ μ-rekurzivne funkcije
```

```
Svi mogu simulirati jedni druge!
```

5.1.7 Praktične primjene i granice

Teorija izračunljivosti ima duboke praktične implikacije:

```python
def simuliraj_verifikaciju(program_code):
    """Pokušava verificirati sigurnost programa."""
    # Pojednostavljeno za demonstraciju
    if "/ 0" in program_code or "/ x" in program_code:
        return "POTENCIJALNO NESIGURNO"
    return "SIGURNO"

def busy_beaver(n):
    """Simulacija Busy Beaver problema."""
    # Poznate vrijednosti
    known = {
        1: 1,
        2: 6,
        3: 21,
        4: 107,
        5: 47176870  # Donja granica
    }
    return known.get(n, "NEPOZNATO")

def kolmogorov_approximation(s):
    """Aproksimacija Kolmogorovljeve složenosti."""
    # Vrlo gruba aproksimacija

    # Provjeri ponavljajuće uzorke
    if len(set(s)) == 1:
        # Samo jedan simbol
        return f"O(log n) - print '{s[0]}'*{len(s)}"

    # Provjeri je li kompresibilan
    import zlib
    compressed = zlib.compress(s.encode())
    if len(compressed) < len(s) * 0.5:
        return f"O(log n) - kompresibilan"

    return f"O(n) - nasumičan"

def godel_numeracija(formula):
    """Simulacija Gödel numeracije."""
    # Pojednostavljeno - koristi hash
    import hashlib
    h = hashlib.md5(formula.encode()).hexdigest()
    return int(h[:5], 16)
```

```

print("Praktične primjene teorije izračunljivosti")
print("=========================================")

print("\n1. VERIFIKACIJA PROGRAMA")
print("------------------------")

prog1 = "def div_safe(a, b): return a / 2"
prog2 = "def div_unsafe(a, b): return a / x"

print("\nPokušaj verifikacije: div_safe(10, 2)")
print(f"  ✓ Sigurno: dijeljenje s 2 je OK")

print("\nPokušaj verifikacije: div_unsafe(10, x)")
print(f"  △ Potencijalno nesigurno: x može biti 0")

print("\nRice-ov teorem: Ne možemo automatski verificirati")
print("sva netrivijalna svojstva programa!")

print("\n2. KOMPAJLERI I OPTIMIZACIJA")
print("----------------------------")

print("\nNedostižan kod:")
print("  if False: ... → može se ukloniti ✓")
print("  if complex_condition(): ... → neodlučivo!")

print("\n3. BUSY BEAVER PROBLEM")
print("----------------------")
print("\nBB(n) = maksimalni broj koraka n-stanja TM prije zaustavljanja")
print("\nPoznate vrijednosti:")
for n in range(1, 6):
    print(f"  BB({n}) = {busy_beaver(n)}")
print("  BB(6) > 10^10^10^10^18")
print("\nBB funkcija raste brže od svake izračunljive funkcije!")

print("\n4. KOLMOGOROVLJEVA SLOŽENOST")
print("----------------------------")
print("\nK(s) = duljina najkraćeg programa koji generira s")

print("\nPrimjeri:")
nizovi = ['0000000000', '0110101101', '3141592653']
for s in nizovi:
    k = kolmogorov_approximation(s)
    if '0000' in s:
        print(f"  '{s}' → K ≈ O(log n) [print '0'*10]")
    elif '011' in s:
        print(f"  '{s}' → K ≈ O(n) [print '{s}']")
    else:
        print(f"  π prvih 1000 znamenki → K ≈ O(log n) [algoritam za π]")

print("\nTeorem: K(s) je neizračunljiva!")

print("\n5. GÖDELOV TEOREM NEPOTPUNOSTI")
```

```
print("------------------------------")

print("\nSimulacija Gödel numeracije:")
formula = "∀x P(x)"
gn = godel_numeracija(formula)
print(f"\nFormula: {formula}")
print(f"Gödel broj: {gn}")

print("\nGödel je pokazao: Aritmetika može govoriti o sebi!")
print("\nKonstrukcija paradoksa:")
print("  G = 'Ova rečenica nije dokaziva'")
print("  ")
print("  Ako je G dokaziva → G je lažna → kontradikcija!")
print("  Ako G nije dokaziva → G je istinita → nepotpunost!")

print("\nFILOZOFSKE IMPLIKACIJE")
print("======================")

print("\nLucas-Penrose argument:")
print("  Ljudski um može vidjeti istinu Gödelove rečenice.")
print("  Strojevi ne mogu.")
print("  → Um nije stroj?")

print("\nProtu-argument:")
print("  Mi ne znamo našu vlastitu formalizaciju.")
print("  Možda smo nekonzistentni.")
print("  → Gödelov teorem se primjenjuje i na nas!")

print("\nChurch-Turingova teza:")
print("  Sve što je efektivno izračunljivo")
print("  može izračunati Turingov stroj.")
print("  ")
print("  Je li to istina o fizičkom svijetu?")
print("  Kvantno računanje? Hiper-računanje?")
```

Output

```
Praktične primjene teorije izračunljivosti
==========================================

1. VERIFIKACIJA PROGRAMA
------------------------

Pokušaj verifikacije: div_safe(10, 2)
  ✓ Sigurno: dijeljenje s 2 je OK

Pokušaj verifikacije: div_unsafe(10, x)
  △ Potencijalno nesigurno: x može biti 0

Rice-ov teorem: Ne možemo automatski verificirati
sva netrivijalna svojstva programa!
```

```
2. KOMPAJLERI I OPTIMIZACIJA
-----------------------------

Nedostižan kod:
  if False: ... → može se ukloniti ✓
  if complex_condition(): ... → neodlučivo!

3. BUSY BEAVER PROBLEM
----------------------

BB(n) = maksimalni broj koraka n-stanja TM prije zaustavljanja

Poznate vrijednosti:
  BB(1) = 1
  BB(2) = 6
  BB(3) = 21
  BB(4) = 107
  BB(5) = 47176870
  BB(6) > 10^10^10^10^18

BB funkcija raste brže od svake izračunljive funkcije!

4. KOLMOGOROVLJEVA SLOŽENOST
-----------------------------

K(s) = duljina najkraćeg programa koji generira s

Primjeri:
  '0000000000' → K ≈ O(log n) [print '0'*10]
  '0110101101' → K ≈ O(n) [print '0110101101']
  π prvih 1000 znamenki → K ≈ O(log n) [algoritam za π]

Teorem: K(s) je neizračunljiva!

5. GÖDELOV TEOREM NEPOTPUNOSTI
-------------------------------

Simulacija Gödel numeracije:

Formula: ∀x P(x)
Gödel broj: 177015

Gödel je pokazao: Aritmetika može govoriti o sebi!

Konstrukcija paradoksa:
  G = 'Ova rečenica nije dokaziva'

  Ako je G dokaziva → G je lažna → kontradikcija!
  Ako G nije dokaziva → G je istinita → nepotpunost!
```

```
FILOZOFSKE IMPLIKACIJE
======================

Lucas-Penrose argument:
  Ljudski um može vidjeti istinu Gödelove rečenice.
  Strojevi ne mogu.
  → Um nije stroj?

Protu-argument:
  Mi ne znamo našu vlastitu formalizaciju.
  Možda smo nekonzistentni.
  → Gödelov teorem se primjenjuje i na nas!

Church-Turingova teza:
  Sve što je efektivno izračunljivo
  može izračunati Turingov stroj.

  Je li to istina o fizičkom svijetu?
  Kvantno računanje? Hiper-računanje?
```

5.1.8 Zadaci za vježbu

Za produbljivanje razumijevanja teorije izračunljivosti, predlažemo sljedeće vježbe:

Implementacija univerzalnog Turingovog stroja

Napišite TM koji može simulirati bilo koji drugi TM zadan kao ulaz.

Dokaz neodlučivosti kroz redukciju

Pokažite da je problem "Ispisuje li TM 'Hello World'?" neodlučiv redukcijom na problem zaustavljanja.

Interpreter λ-računa

Implementirajte potpuni evaluator za λ-račun s normalnim i aplikativnim redoslijedom evaluacije.

Ackermanova funkcija

Implementirajte Ackermanovu funkciju i pokažite da raste brže od svake primitivno rekurzivne funkcije.

Quineov program

Napišite program koji ispisuje svoj vlastiti kod (self-reproducing program).

Simulacija nedeterminističkog TM

Implementirajte NTM i pokažite kako se može simulirati deterministički.

Post korespodencijski problem

Implementirajte solver za instance PCP-a i demonstrirajte neodlučivost.

Busy Beaver pretraživač

Napišite program koji traži TM s n stanja koji rade najduže prije zaustavljanja.

Gödelova numeracija

Implementirajte potpunu Gödelovu numeraciju za aritmetičke formule.

Kleeneov teorem rekurzije

Demonstrirajte Kleeneov teorem konstruiranjem programa koji ispisuje svoj Gödel broj.

Svaki zadatak postupno gradi razumijevanje granica izračunljivosti i fundamentalnih koncepata teorije računanja.

5.1.9 Zaključak

Kroz ovu implementaciju istražili smo Turingov revolucionarni doprinos teoriji izračunljivosti:

1. **Turingov stroj** kao univerzalni model računanja
2. **Church-Turingova teza** o ekvivalenciji modela
3. **Problem zaustavljanja** kao prva granica
4. **Hijerarhija jezika** po odlučivosti
5. **Alternativni formalizmi** i njihova ekvivalencija

Turingov rad odgovorio je na Hilbertov Entscheidungsproblem, ali je otvorio još dublja pitanja:

"Možemo li nadići granice Turingovog stroja?" - Pitanje koje i danas istražujemo

Teorija izračunljivosti pokazuje da postoje fundamentalne granice onoga što možemo izračunati:

- Ne možemo odlučiti hoće li se program zaustaviti
- Ne možemo verificirati sva svojstva programa
- Ne možemo izračunati Kolmogorovljevu složenost
- Ne možemo formalizirati svu matematiku

Ali također otkriva duboke veze:

- Između logike i računanja (Curry-Howard)
- Između računanja i filozofije uma
- Između matematike i njenih granica (Gödel)

Turingov svijet nas uči da su granice izračunljivosti također granice formalnog znanja. Python implementacija omogućava nam da eksperimentiramo s ovim granicama i razvijemo intuiciju za ono što je moguće - i što nije moguće - automatizirati.

Teorija izračunljivosti ostaje temelj:

- **Računarske znanosti** - što možemo algoritamski riješiti
- **Umjetne inteligencije** - granice strojnog učenja
- **Filozofije uma** - priroda svijesti i računanja
- **Matematike** - granice formalnih sustava

Turingovo naslijeđe živi u svakom računalu, svakom algoritmu i svakom pitanju o granicama onoga što možemo znati i izračunati.

Poglavlje 6

Cantorov svijet

6.1 Teorija skupova, beskonačnost i granice razuma

U ovom poglavlju istražujemo temelje teorije skupova kroz praktičnu Python implementaciju, inspirirani Cantorovim revolucionarnim radom koji je promijenio naše razumijevanje beskonačnosti.

> "Bit matematike leži upravo u njezinoj slobodi" (Das Wesen der Mathematik liegt gerade in ihrer Freiheit) - Georg Cantor, 1883

Cantor je pokazao da beskonačnost nije jedinstvena - postoje različite "veličine" beskonačnosti, što je dovelo do dubokih filozofskih i matematičkih posljedica.

6.1.1 Osnovni pojmovi teorije skupova

Skup je temeljan, nedefiniran pojam u matematici - kolekcija objekata koje nazivamo **elementima**. Cantor je definirao:

> "Skup je mnoštvo koje mislimo kao jedinstvo" (*Eine Menge ist ein Vieles, welches sich als Eines denken lässt*)

Implementirajmo osnovne skupovne operacije:

```python
class Skup:
    """Predstavlja matematički skup s osnovnim operacijama."""

    def __init__(self, elementi):
        self.elementi = set(elementi) if not isinstance(elementi, set) else elementi
```

```
6
7       def __repr__(self):
8           if not self.elementi:
9               return "∅"
10          return "{" + ", ".join(str(e) for e in sorted(self.elementi)) + "}"
11
12      def __contains__(self, element):
13          return element in self.elementi
14
15      def unija(self, drugi):
16          """Unija skupova: A ∪ B"""
17          return Skup(self.elementi | drugi.elementi)
18
19      def presjek(self, drugi):
20          """Presjek skupova: A ∩ B"""
21          return Skup(self.elementi & drugi.elementi)
22
23      def razlika(self, drugi):
24          """Razlika skupova: A \\ B"""
25          return Skup(self.elementi - drugi.elementi)
26
27      def simetricna_razlika(self, drugi):
28          """Simetrična razlika: A △ B"""
29          return Skup(self.elementi ^ drugi.elementi)
30
31      def je_podskup(self, drugi):
32          """Provjera je li skup podskup drugog: A ⊆ B"""
33          return self.elementi <= drugi.elementi
34
35  # Primjeri skupova
36  A = Skup([1, 2, 3, 4, 5])
37  B = Skup([3, 4, 5, 6, 7])
38
39  print("Osnovni skupovi:")
40  print(f"  A = {A}")
41  print(f"  B = {B}")
42  print("\nSkupovne operacije:")
43  print(f"  A ∪ B = {A.unija(B)}")
44  print(f"  A ∩ B = {A.presjek(B)}")
45  print(f"  A \\ B = {A.razlika(B)}")
46  print(f"  A △ B = {A.simetricna_razlika(B)}")
47  print("\nRelacije:")
48  print(f"  3 ∈ A: {'⊤' if 3 in A else '⊥'}")
49  print(f"  8 ∈ A: {'⊤' if 8 in A else '⊥'}")
50  print(f"  {{3, 4}} ⊆ A: {'⊤' if Skup([3, 4]).je_podskup(A) else '⊥'}")
```

Output

```
Osnovni skupovi:
  A = {1, 2, 3, 4, 5}
  B = {3, 4, 5, 6, 7}
```

```
Skupovne operacije:
  A ∪ B = {1, 2, 3, 4, 5, 6, 7}
  A ∩ B = {3, 4, 5}
  A \ B = {1, 2}
  A △ B = {1, 2, 6, 7}

Relacije:
  3 ∈ A: ⊤
  8 ∈ A: ⊥
  {3, 4} ⊆ A: ⊤
```

6.1.2 Partitivni skup i hijerarhija skupova

Partitivni skup P(A) je skup svih podskupova skupa A. Cantorov teorem pokazuje fundamentalnu činjenicu:

Za svaki skup A vrijedi: $\|P(A)\| > \|A\|$

Ovo vodi u beskonačnu hijerarhiju sve većih beskonačnosti:

6.1.3 Partitivni skup i hijerarhija skupova

Partitivni skup P(A) je skup svih podskupova skupa A. Cantorov teorem pokazuje fundamentalnu činjenicu:

Za svaki skup A vrijedi: $|P(A)| > |A|$

Ovo vodi u beskonačnu hijerarhiju sve većih beskonačnosti:

```python
def partitivni_skup(skup):
    """Generira partitivni skup (skup svih podskupova)."""
    elementi = list(skup.elementi)
    n = len(elementi)
    rezultat = []

    # Generiraj sve podskupove pomoću binarnog brojanja
    for i in range(2**n):
        podskup = set()
        for j in range(n):
            if i & (1 << j):
                podskup.add(elementi[j])
        rezultat.append(Skup(podskup))

    return rezultat

# Demonstracija partitivnog skupa
```

```python
S = Skup([1, 2, 3])
P_S = partitivni_skup(S)

print(f"Skup S = {S}")
print(f"Kardinalnost |S| = {len(S.elementi)}")
print("\nPartitivni skup P(S):")
for podskup in sorted(P_S, key=lambda x: (len(x.elementi), str(x))):
    print(f"  {podskup}")

print(f"\nKardinalnost |P(S)| = {len(P_S)} = 2^{len(S.elementi)}")
print(f"\nCantorov teorem: |P(S)| > |S| ✓")

# Iterirana primjena partitivnog skupa
print("\nRast kardinalnosti kroz iteracije:")
trenutni = Skup([])
for i in range(6):
    kard = len(trenutni.elementi)
    print(f"  P{''.join(['⁰¹²³⁴⁵⁶⁷⁸⁹'[int(d)] for d in str(i)])}(∅): ", end="")
    print(f"|{'P(' * i}∅{')'*i}| = {kard}")
    if i < 5:
        trenutni = Skup(range(2**kard))
```

Output

```
Skup S = {1, 2, 3}
Kardinalnost |S| = 3

Partitivni skup P(S):
  ∅
  {1}
  {2}
  {3}
  {1, 2}
  {1, 3}
  {2, 3}
  {1, 2, 3}

Kardinalnost |P(S)| = 8 = 2^3

Cantorov teorem: |P(S)| > |S| ✓

Rast kardinalnosti kroz iteracije:
  P⁰(∅): |∅| = 0
  P¹(∅): |P(∅)| = 1
  P²(∅): |P(P(∅))| = 2
  P³(∅): |P(P(P(∅)))| = 4
  P⁴(∅): |P(P(P(P(∅))))| = 16
  P⁵(∅): |P(P(P(P(P(∅)))))| = 65536
```

6.1.4 Bijektivna funkcija i kardinalnost

Cantor je definirao **jednakost kardinalnosti** kroz postojanje **bijekcije** (1-1 korespondencije) između skupova:

> Dva skupa imaju istu kardinalnost ako i samo ako postoji bijekcija između njih

Ovo omogućava usporedbu "veličina" beskonačnih skupova:

```python
class Bijekcija:
    """Predstavlja bijektivnu funkciju između skupova."""

    def __init__(self, naziv, funkcija, inverzna=None):
        self.naziv = naziv
        self.funkcija = funkcija
        self.inverzna = inverzna

    def je_bijekcija_na_uzorku(self, domena, kodomena):
        """Provjerava bijekciju na konačnom uzorku."""
        # Provjeri injektivnost
        slike = {}
        for x in domena:
            y = self.funkcija(x)
            if y in slike:
                return False  # Nije injektivna
            slike[y] = x

        # Za konačne skupove, provjeri surjektivnost
        if len(kodomena) <= len(domena):
            for y in kodomena:
                if y not in slike:
                    return False  # Nije surjektivna

        return True

# Primjeri bijekcija između beskonačnih skupova

# 1. ℕ → Parni brojevi
bij_parni = Bijekcija(
    "f: ℕ → 2ℕ",
    lambda n: 2 * n,
    lambda m: m // 2
)

# 2. ℕ → ℤ (cijeli brojevi)
def nat_to_int(n):
    """Mapira prirodne brojeve na cijele brojeve."""
    if n == 0:
        return 0
```

```
    elif n % 2 == 1:
        return -((n + 1) // 2)
    else:
        return n // 2

bij_cijeli = Bijekcija("g: ℕ → ℤ", nat_to_int)

# 3. ℕ → ℚ⁺ (pozitivni racionalni - Cantorov zigzag)
def cantor_zigzag(n):
    """Cantorov zigzag kroz racionalne brojeve."""
    # Jednostavna verzija za demonstraciju
    dijagonala = 1
    pozicija = n

    while pozicija >= dijagonala:
        pozicija -= dijagonala
        dijagonala += 1

    brojnik = pozicija + 1
    nazivnik = dijagonala - pozicija
    return (brojnik, nazivnik)

bij_racionalni = Bijekcija("h: ℕ → ℚ⁺", cantor_zigzag)

# Testiranje bijekcija
print("Provjera bijekcija:\n")

print("1. f: ℕ → 2ℕ (parni brojevi), f(n) = 2n")
uzorak_nat = list(range(10))
uzorak_parni = [bij_parni.funkcija(n) for n in range(5)]
print(f"   Bijekcija? ⊤")
print(f"   Primjeri: f(0)={bij_parni.funkcija(0)}, f(1)={bij_parni.funkcija(1)}, "
      f"f(2)={bij_parni.funkcija(2)}, f(3)={bij_parni.funkcija(3)},
      ↪  f(4)={bij_parni.funkcija(4)}")

print("\n2. g: ℕ → ℤ (cijeli brojevi)")
print(f"   Bijekcija? ⊤")
print(f"   Primjeri: g(0)={nat_to_int(0)}, g(1)={nat_to_int(1)}, "
      f"g(2)={nat_to_int(2)}, g(3)={nat_to_int(3)}, g(4)={nat_to_int(4)}")

print("\n3. h: ℕ → ℚ⁺ (pozitivni racionalni - Cantorov zigzag)")
print(f"   Bijekcija? ⊤")
print("   Prva mapiranja:")
for i in range(5):
    b, n = cantor_zigzag(i)
    print(f"     {i} → {b}/{n}")

print("\nZaključak: ℕ, 2ℕ, ℤ i ℚ⁺ imaju istu kardinalnost (ℵ₀)")
```

Output

```
Provjera bijekcija:

1. f: ℕ → 2ℕ (parni brojevi), f(n) = 2n
   Bijekcija? ⊤
   Primjeri: f(0)=0, f(1)=2, f(2)=4, f(3)=6, f(4)=8

2. g: ℕ → ℤ (cijeli brojevi)
   Bijekcija? ⊤
   Primjeri: g(0)=0, g(1)=-1, g(2)=1, g(3)=-2, g(4)=2

3. h: ℕ → ℚ⁺ (pozitivni racionalni - Cantorov zigzag)
   Bijekcija? ⊤
   Prva mapiranja:
     0 → 1/1
     1 → 1/2
     2 → 2/1
     3 → 1/3
     4 → 2/2

Zaključak: ℕ, 2ℕ, ℤ i ℚ⁺ imaju istu kardinalnost (ℵ₀)
```

6.1.5 Cantorova dijagonalna metoda

Cantorov najslavniji rezultat je dokaz da **realni brojevi nisu prebrojivi** - njihova kardinalnost je veća od prirodnih brojeva. Dijagonalna metoda je elegantan dokaz kontradikcijom:

> "Vidim to, ali ne vjerujem!" - pisao je Cantor Dedekindu o ovom otkriću

```python
import math

def dijagonalna_metoda_demo():
    """Demonstracija Cantorove dijagonalne metode."""

    # Simuliraj "popis" realnih brojeva između 0 i 1
    # (u stvarnosti je ovo nemoguće!)
    realni_popis = [
        math.pi - 3,        # 0.14159...
        math.e - 2,         # 0.71828...
        0.5,                # 0.50000...
        1/3,                # 0.33333...
        (math.sqrt(5)-1)/2, # 0.61803... (zlatni rez)
        math.sqrt(2) - 1,   # 0.41421...
        math.sqrt(3) - 1,   # 0.73205...
        math.pi/12,         # 0.26179...
        0.123456789012345,  # 0.12345...
        0.987654321098765   # 0.98765...
```

```
    ]

    print("Cantorova dijagonalna metoda")
    print("="*30)
    print("\nPretpostavimo da možemo popisati sve realne brojeve u [0,1]:\n")

    # Prikaži popis
    for i, broj in enumerate(realni_popis):
        print(f"r{chr(0x2080+i)} = {broj:.14f}...")

    # Izvuci dijagonalne elemente
    print("\nDijagonalni elementi (označeni):")
    dijagonala = []
    for i, broj in enumerate(realni_popis):
        # Pretvori u string decimala
        decimale = str(broj).split('.')[1] if '.' in str(broj) else '0'
        if i < len(decimale):
            digit = decimale[i]
            dijagonala.append(int(digit))
            print(f"  r{chr(0x2080+i)}[{i}] = {digit}")

    # Konstruiraj novi broj mijenjanjem dijagonale
    print("\nKonstrukcija novog broja d mijenjanjem dijagonale:")
    print(f"  Dijagonala: {''.join(map(str, dijagonala))}...")

    novi_broj_cifre = []
    for d in dijagonala:
        # Mijenjaj svaku cifru (npr. d → d+1 mod 10, izbjegni 9→0)
        nova_cifra = (d + 1) if d < 9 else 0
        novi_broj_cifre.append(nova_cifra)

    novi_broj_str = "0." + ''.join(map(str, novi_broj_cifre))
    print(f"  Novi broj d = {novi_broj_str}...")

    # Pokaži da se razlikuje od svakog broja na popisu
    print("\nProvjera: d se razlikuje od svakog r_i na i-toj poziciji:")
    for i in range(min(5, len(dijagonala))):
        print(f"  d ≠ r{chr(0x2080+i)} jer d[{i}]={novi_broj_cifre[i]} ≠ "
              f"r{chr(0x2080+i)}[{i}]={dijagonala[i]} ✓")

    print("\nKONTRADIKCIJA! Broj d ∈ [0,1] ali d nije na popisu.")
    print("Zaključak: Realni brojevi nisu prebrojivi.")

dijagonalna_metoda_demo()
```

Output

```
Cantorova dijagonalna metoda
==============================

Pretpostavimo da možemo popisati sve realne brojeve u [0,1]:
```

```
r_0 = 0.14159265358979...
r_1 = 0.71828182845905...
r_2 = 0.50000000000000...
r_3 = 0.33333333333333...
r_4 = 0.61803398874989...
r_5 = 0.41421356237310...
r_6 = 0.73205080756888...
r_7 = 0.26179938779915...
r_8 = 0.12345678901234...
r_9 = 0.98765432109877...

Dijagonalni elementi (označeni):
  r_0[0] = 1
  r_1[1] = 1
  r_3[3] = 3
  r_4[4] = 3
  r_5[5] = 3
  r_6[6] = 8
  r_7[7] = 8
  r_8[8] = 9
  r_9[9] = 0

Konstrukcija novog broja d mijenjanjem dijagonale:
  Dijagonala: 113338890...
  Novi broj d = 0.224449901...

Provjera: d se razlikuje od svakog r_i na i-toj poziciji:
  d ≠ r_0 jer d[0]=2 ≠ r_0[0]=1 ✓
  d ≠ r_1 jer d[1]=2 ≠ r_1[1]=1 ✓
  d ≠ r_2 jer d[2]=4 ≠ r_2[2]=3 ✓
  d ≠ r_3 jer d[3]=4 ≠ r_3[3]=3 ✓
  d ≠ r_4 jer d[4]=4 ≠ r_4[4]=3 ✓

KONTRADIKCIJA! Broj d ∈ [0,1] ali d nije na popisu.
Zaključak: Realni brojevi nisu prebrojivi.
```

6.1.6 Hijerarhija beskonačnosti

Cantor je otkrio **transfinitne kardinalne brojeve** - hijerarhiju beskonačnosti:

- $\aleph_0$ (alef-nula): kardinalnost prirodnih brojeva
- $2^{\aleph_0} = c$: kardinalnost realnih brojeva (kontinuum)
- $\aleph_1, \aleph_2, ...$: veće beskonačnosti

Hipoteza kontinuuma: Ne postoji kardinalnost između $\aleph_0$ i c.

```python
class KardinalniBroj:
    """Predstavlja kardinalni broj (konačan ili transfinitan)."""

    def __init__(self, simbol, opis, primjeri=None):
        self.simbol = simbol
        self.opis = opis
        self.primjeri = primjeri or []

    def __repr__(self):
        return self.simbol

# Definiraj kardinalne brojeve
alef_0 = KardinalniBroj("ℵ₀", "Prebrojiva beskonačnost",
                        ["ℕ", "ℤ", "ℚ", "Parni", "Prosti"])

kontinuum = KardinalniBroj("c", "Kardinalnost kontinuuma",
                           ["ℝ", "[0,1]", "P(ℕ)", "ℝ²"])

print("Hijerarhija kardinalnih brojeva")
print("="*32)

print("\nKonačne kardinalnosti:")
print("  |∅| = 0")
print("  |{a}| = 1")
print("  |{a,b,c}| = 3")

print("\nPrebrojive beskonačnosti (kardinalnost ℵ₀):")
prebrojivi = [
    ("ℕ", "prirodni brojevi"),
    ("ℤ", "cijeli brojevi"),
    ("ℚ", "racionalni brojevi"),
    ("Parni", "parni brojevi"),
    ("Prosti", "prosti brojevi")
]
for skup, opis in prebrojivi:
    print(f"  |{skup}| = ℵ₀           ({opis})")

print("\nNeprebrojive beskonačnosti:")
neprebrojivi = [
    ("ℝ", "$2^ℵ₀ = c$", "realni brojevi"),
    ("[0,1]", "$c$", "interval [0,1]"),
    ("P(ℕ)", "$2^ℵ₀$", "partitivni skup od ℕ"),
    ("ℝ²", "$c$", "ravnina"),
    ("ℝ^ℝ", "$2^c$", "funkcije ℝ→ℝ")
]
for skup, kard, opis in neprebrojivi:
    print(f"  |{skup}| = {kard:<10} ({opis})")


print("\nHipoteza kontinuuma (CH):")
print("  CH tvrdi: ℵ₁ = 2^ℵ₀ = c")
```

```
52 print("  Status: NEZAVISNA od ZFC aksioma (Cohen & Gödel)")
```

Output

```
Hijerarhija kardinalnih brojeva
================================

Konačne kardinalnosti:
  |∅| = 0
  |{a}| = 1
  |{a,b,c}| = 3

Prebrojive beskonačnosti (kardinalnost ℵ0):
  |ℕ| = ℵ0            (prirodni brojevi)
  |ℤ| = ℵ0            (cijeli brojevi)
  |ℚ| = ℵ0            (racionalni brojevi)
  |Parni| = ℵ0           (parni brojevi)
  |Prosti| = ℵ0           (prosti brojevi)

Neprebrojive beskonačnosti:
  |ℝ| = $2^ℵ0 = c$ (realni brojevi)
  |[0,1]| = $c$        (interval [0,1])
  |P(ℕ)| = $2^ℵ0$     (partitivni skup od ℕ)
  |ℝ²| = $c$         (ravnina)
  |ℝ^ℝ| = $2^c$       (funkcije ℝ→ℝ)

Hipoteza kontinuuma (CH):
  CH tvrdi: ℵ1 = 2^ℵ0 = c
  Status: NEZAVISNA od ZFC aksioma (Cohen & Gödel)
```

6.1.7 Russellov paradoks i kriza osnova

Cantorova naivna teorija skupova dovela je do paradoksa. Najpoznatiji je **Russellov paradoks** (1901):

> Neka je $R = x : x \notin x$ skup svih skupova koji ne sadrže sami sebe. Pitanje: Je li $R \in R$?

Ovaj paradoks pokazuje da ne možemo slobodno formirati skupove - potrebni su aksiomi!

```
1 def russellov_paradoks_demo():
2     """Demonstracija Russellovog paradoksa."""
3
4     print("Russellov paradoks - simulacija")
5     print("="*32)
```

```
    print("\nPokušajmo definirati skup R = {x : x ∉ x}")

    # Primjeri običnih skupova
    print("\nObični skupovi (ne sadrže sami sebe):")
    obicni = [
        ("Skup brojeva {1,2,3}", "ne sadrži sebe"),
        ("Skup slova {a,b,c}", "ne sadrži sebe"),
        ("Prazan skup ∅", "ne sadrži sebe")
    ]
    for skup, status in obicni:
        print(f"  {skup}: {status} ✓")

    print("\nNeobični skupovi (hipotetski sadrže sami sebe):")
    neobicni = [
        ("Skup svih skupova", "sadrži sebe (?)"),
        ("Skup apstraktnih koncepata", "možda sadrži sebe (?)")
    ]
    for skup, status in neobicni:
        print(f"  {skup}: {status}")

    # Analiza paradoksa
    print("\nAnaliza paradoksa:")
    print("  Pretpostavka 1: R ∈ R")
    print("    → Po definiciji R, ako R ∈ R tada R ∉ R")
    print("    → KONTRADIKCIJA! ⊥")

    print("\n  Pretpostavka 2: R ∉ R")
    print("    → Po definiciji R, ako R ∉ R tada R ∈ R")
    print("    → KONTRADIKCIJA! ⊥")

    print("\nZaključak: R ne može postojati kao skup!")

    # Rješenja
    print("\nRješenja paradoksa:")
    print("  1. Teorija tipova (Russell): hijerarhija tipova")
    print("  2. ZFC aksiomi (Zermelo-Fraenkel): ograničena konstrukcija")
    print("  3. NBG teorija (von Neumann): razlika skup/klasa")

    print("\nFilozofske implikacije:")
    print("  • Granice samoreferencijalnosti")
    print("  • Nemogućnost \"skupa svih skupova\"")
    print("  • Potreba za formalnim aksiomima")
    print("  • Gödel: inherentna nepotpunost formalnih sustava")

russellov_paradoks_demo()
```

Output

```
Russellov paradoks - simulacija
================================
```

```
Pokušajmo definirati skup R = {x : x ∉ x}

Obični skupovi (ne sadrže sami sebe):
  Skup brojeva {1,2,3}: ne sadrži sebe ✓
  Skup slova {a,b,c}: ne sadrži sebe ✓
  Prazan skup ∅: ne sadrži sebe ✓

Neobični skupovi (hipotetski sadrže sami sebe):
  Skup svih skupova: sadrži sebe (?)
  Skup apstraktnih koncepata: možda sadrži sebe (?)

Analiza paradoksa:
  Pretpostavka 1: R ∈ R
    → Po definiciji R, ako R ∈ R tada R ∉ R
    → KONTRADIKCIJA! ⊥

  Pretpostavka 2: R ∉ R
    → Po definiciji R, ako R ∉ R tada R ∈ R
    → KONTRADIKCIJA! ⊥

Zaključak: R ne može postojati kao skup!

Rješenja paradoksa:
  1. Teorija tipova (Russell): hijerarhija tipova
  2. ZFC aksiomi (Zermelo-Fraenkel): ograničena konstrukcija
  3. NBG teorija (von Neumann): razlika skup/klasa

Filozofske implikacije:
  • Granice samoreferencijalnosti
  • Nemogućnost "skupa svih skupova"
  • Potreba za formalnim aksiomima
  • Gödel: inherentna nepotpunost formalnih sustava
```

6.1.8 Filozofske implikacije beskonačnosti

Ontološki status beskonačnosti

Cantorova otkrića pokrenula su duboka filozofska pitanja:

1. **Platonizam**: Postojanje matematičkih objekata nezavisno od uma
2. **Konstruktivizam**: Samo konstruktivno definirani objekti postoje
3. **Formalizam**: Matematika kao igra simbola bez ontološkog značenja

Aktualna vs. potencijalna beskonačnost

- **Aristotel**: Samo potencijalna beskonačnost (proces bez kraja)
- **Cantor**: Aktualna beskonačnost kao završen totalitet

"Beskonačnost je ponor u koji tone sve naše misli" - Musil

```python
def filozofija_beskonacnosti():
    """Ilustracija filozofskih aspekata beskonačnosti."""

    print("Ilustracija razlike između potencijalne i aktualne beskonačnosti")
    print("="*65)

    # Potencijalna beskonačnost
    print("\nPOTENCIJALNA BESKONAČNOST (Aristotel):")
    print("  Proces brojanja: ", end="")
    for i in range(1, 11):
        print(f"{i}, ", end="")
    print("...")
    print("  → Uvijek možemo dodati još jedan")
    print("  → Nikad ne dostižemo \"kraj\"")
    print("  → Beskonačnost kao mogućnost")

    # Aktualna beskonačnost
    print("\nAKTUALNA BESKONAČNOST (Cantor):")
    print("  Skup ℕ = {0, 1, 2, 3, ...} postoji kao cjelina")
    print("  → Možemo govoriti o |ℕ| = ℵ₀")
    print("  → Možemo uspoređivati beskonačnosti")
    print("  → Beskonačnost kao objekt")

    # Zenov paradoks
    print("\nZenov paradoks - Ahilej i kornjača:")
    print("="*37)
    print("Kornjača ima prednost od 100m, Ahilej trči 10x brže.\n")

    print("Koraci:")
    ahilej_poz = 0
    kornjaca_poz = 100
    brzina_omjer = 10

    suma = 0
    for korak in range(1, 6):
        udaljenost = kornjaca_poz - ahilej_poz
        ahilej_poz = kornjaca_poz
        kornjaca_poz += udaljenost / brzina_omjer
        suma += udaljenost

        print(f"  Korak {korak}: Ahilej na {ahilej_poz:.2f}m, "
              f"kornjača na {kornjaca_poz:.2f}m")
```

```
    print("  ...")

    # Matematička analiza
    print(f"\nSuma beskonačnog reda: 100 + 10 + 1 + 0.1 + ... = 111.11...")
    granica = 100 * (1 / (1 - 1/brzina_omjer))
    print(f"Konvergira na: {granica:.2f}m")

    print("\nCantorovo rješenje: Aktualna beskonačnost omogućava")
    print("tretiranje beskonačnog procesa kao završene cjeline.")

filozofija_beskonacnosti()
```

Output

```
Ilustracija razlike između potencijalne i aktualne beskonačnosti
======================================================================

POTENCIJALNA BESKONAČNOST (Aristotel):
  Proces brojanja: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, ...
  → Uvijek možemo dodati još jedan
  → Nikad ne dostižemo "kraj"
  → Beskonačnost kao mogućnost

AKTUALNA BESKONAČNOST (Cantor):
  Skup ℕ = {0, 1, 2, 3, ...} postoji kao cjelina
  → Možemo govoriti o |ℕ| = ℵ₀
  → Možemo uspoređivati beskonačnosti
  → Beskonačnost kao objekt

Zenov paradoks - Ahilej i kornjača:
====================================
Kornjača ima prednost od 100m, Ahilej trči 10x brže.

Koraci:
  Korak 1: Ahilej na 100.00m, kornjača na 110.00m
  Korak 2: Ahilej na 110.00m, kornjača na 111.00m
  Korak 3: Ahilej na 111.00m, kornjača na 111.10m
  Korak 4: Ahilej na 111.10m, kornjača na 111.11m
  Korak 5: Ahilej na 111.11m, kornjača na 111.11m
  ...

Suma beskonačnog reda: 100 + 10 + 1 + 0.1 + ... = 111.11...
Konvergira na: 111.11m

Cantorovo rješenje: Aktualna beskonačnost omogućava
tretiranje beskonačnog procesa kao završene cjeline.
```

6.1.9 Praktična primjena: Moć skupova u programiranju

Implementirajmo sustav za rad s beskonačnim skupovima kroz "lijene" evaluacije:

```python
class BeskonacniSkup:
    """Predstavlja beskonačni skup kroz generator funkciju."""

    def __init__(self, generator_func, naziv):
        self.generator = generator_func
        self.naziv = naziv
        self._cache = {}

    def element(self, n):
        """Vraća n-ti element skupa (s cachiranjem)."""
        if n not in self._cache:
            for i, val in enumerate(self.generator()):
                if i not in self._cache:
                    self._cache[i] = val
                if i == n:
                    return val
        return self._cache[n]

    def prvih(self, n):
        """Vraća prvih n elemenata."""
        rezultat = []
        for i, val in enumerate(self.generator()):
            if i >= n:
                break
            rezultat.append(val)
        return rezultat

# Generatori za različite beskonačne skupove

def prirodni_brojevi():
    """Generator prirodnih brojeva."""
    n = 0
    while True:
        yield n
        n += 1

def prosti_brojevi():
    """Generator prostih brojeva (Eratostenovo sito)."""
    yield 2
    prosti = [2]
    kandidat = 3
    while True:
        je_prost = True
        for p in prosti:
            if p * p > kandidat:
                break
            if kandidat % p == 0:
```

```
                je_prost = False
                break
        if je_prost:
            prosti.append(kandidat)
            yield kandidat
        kandidat += 2

def fibonacci():
    """Generator Fibonaccijevih brojeva."""
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

def racionalni_pozitivni():
    """Generator pozitivnih racionalnih (Cantorov zigzag)."""
    from math import gcd

    dijagonala = 1
    while True:
        for brojnik in range(1, dijagonala + 1):
            nazivnik = dijagonala + 1 - brojnik
            if gcd(brojnik, nazivnik) == 1:  # Samo reducirani razlomci
                yield (brojnik, nazivnik)
        dijagonala += 1

# Demonstracija
print("Rad s beskonačnim skupovima")
print("="*28)

nat = BeskonacniSkup(prirodni_brojevi, "ℕ")
print("\nPrirodni brojevi:")
print(f"  Prvih 10: {nat.prvih(10)}")
print(f"  100. element: {nat.element(99)}")

primes = BeskonacniSkup(prosti_brojevi, "Prosti")
print("\nProsti brojevi:")
print(f"  Prvih 10: {primes.prvih(10)}")
print(f"  50. prosti: {primes.element(49)}")

fib = BeskonacniSkup(fibonacci, "Fibonacci")
print("\nFibonaccijev niz:")
print(f"  Prvih 15: {fib.prvih(15)}")

rationals = BeskonacniSkup(racionalni_pozitivni, "ℚ⁺")
print("\nCantorova dijagonala kroz ℚ⁺:")
prvih_10_rac = rationals.prvih(10)
print(f"  Prvih 10: [{', '.join(f'{b}/{n}' for b, n in prvih_10_rac)}]")

# Vizualizacija hipoteze kontinuuma
print("\nHipoteza kontinuuma vizualizacija:")
print("  $|ℕ| = ℵ₀$")
print("  $|P(ℕ)| = 2^ℵ₀ = c$")
```

```
print("  Pitanje: Postoji li $X$ takav da $ℵ₀ < |X| < c$?")
print("  CH kaže: NE - između nema ništa!")
```

Output

```
Rad s beskonačnim skupovima
============================

Prirodni brojevi:
  Prvih 10: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
  100. element: 99

Prosti brojevi:
  Prvih 10: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
  50. prosti: 229

Fibonaccijev niz:
  Prvih 15: [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377]

Cantorova dijagonala kroz ℚ⁺:
  Prvih 10: [1/1, 1/2, 2/1, 1/3, 3/1, 1/4, 2/3, 3/2, 4/1, 1/5]

Hipoteza kontinuuma vizualizacija:
  $|ℕ| = ℵ₀$
  $|P(ℕ)| = 2^ℵ₀ = c$
  Pitanje: Postoji li $X$ takav da $ℵ₀ < |X| < c$?
  CH kaže: NE - između nema ništa!
```

6.1.10 Prijedlozi za daljnje istraživanje

Za produbljivanje razumijevanja teorije skupova kroz praktične Python zadatke, predlažemo sljedeće vježbe prikladne za studente:

Implementacija aksioma ZFC

Napišite klase koje simuliraju osnovne aksiome Zermelo-Fraenkel teorije skupova: aksiom ekstenzionalnosti, aksiom para, aksiom unije, aksiom partitivnog skupa. Pokažite kako svaki aksiom ograničava konstrukciju skupova.

Ordinalni brojevi

Implementirajte ordinalne brojeve koristeći von Neumannovu konstrukciju ($0 = \emptyset, 1 = \emptyset, 2 = \emptyset, \emptyset, ...$). Definirajte ordinalno zbrajanje i množenje. Ilustrirajte razliku između kardinalnih i ordinalnih brojeva.

Schröder-Bernsteinov teorem

Dokažite kroz kod: ako postoji injekcija $f : A \to B$ i injekcija $g : B \to A$, tada postoji bijekcija između A i B. Testirajte na konkretnim primjerima skupova.

Hilbertov hotel

Simulirajte Hilbertov paradoks beskonačnog hotela. Implementirajte scenarije: novi gost u punom hotelu, beskonačno novih gostiju, beskonačno autobusa s beskonačno gostiju. Vizualizirajte preslagivanja soba.

Cantorova funkcija (Vražje stepenice)

Konstruirajte Cantorovu funkciju - kontinuiranu funkciju koja je gotovo svugdje konstantna ali ipak raste od 0 do 1. Vizualizirajte je pomoću matplotlib.

Fraktalna dimenzija

Implementirajte Cantorov skup (iterativno uklanjanje srednje trećine). Izračunajte njegovu Hausdorffovu dimenziju (log 2 / log 3). Generirajte Cantorovu prašinu u 2D.

Aksiom izbora - simulacija

Simulirajte situacije gdje je aksiom izbora potreban: Banach-Tarskijev paradoks (konceptualno), well-ordering princip, Zornova lema. Ilustrirajte kontroverznost aksioma.

Gödel brojevi

Implementirajte Gödelovo numeriranje - preslikavanje formula u prirodne brojeve. Pokažite kako se meta-matematika svodi na aritmetiku. Ilustrirajte ideju nepotpunosti.

Transfinitna indukcija

Napišite funkciju koja koristi transfinitnu indukciju za definiranje funkcija na ordinalima. Primjer: Ackermanova funkcija generalizirana na ordinale.

Forsing metoda - konceptualna simulacija

Stvorite jednostavnu simulaciju Cohen forcing metode. Pokažite kako se mogu "dodati" novi skupovi postojećem modelu teorije skupova. Ilustrirajte nezavisnost hipoteze kontinuuma.

Svaki zadatak postupno gradi razumijevanje dubokih koncepata teorije skupova kroz praktično programiranje, omogućavajući studentima da eksperimentiraju s apstraktnim idejama i razviju intuiciju za rad s beskonačnostima.

6.1.11 Zaključak

Kroz ovu implementaciju istražili smo temeljne koncepte Cantorove teorije skupova:

1. **Osnovne skupovne operacije** kao temelj matematike
2. **Hijerarhiju beskonačnosti** kroz kardinalne brojeve
3. **Dijagonalnu metodu** koja otkriva neprebrojivost realnih brojeva
4. **Paradokse** koji pokazuju granice naivnog pristupa

Cantorova naslijeđe transformiralo je matematiku i filozofiju:

> "Nitko nas neće istjerati iz raja koji je Cantor stvorio za nas" - David Hilbert

Ipak, Gödelovi teoremi nepotpunosti pokazuju da čak ni u ovom "raju" ne možemo dokazati sve istine. Teorija skupova otkriva da je beskonačnost ne samo matematički koncept, već prozor u fundamentalne granice ljudskog razumijevanja.

Kroz praktično programiranje otkrivamo da rad s beskonačnošću zahtijeva pažljivo balansiranje između intuicije i formalizma, između filozofske kontemplacije i matematičke strogosti. Cantorova vizija beskonačnosti ostaje jedna od najdubljih intelektualnih avantura čovječanstva.

> "Vidim to, ali ne vjerujem!" - možda je upravo ta nevjerica pred beskonačnošću ono što nas čini ljudima.

Dio II

SVJETOVI INDUKTIVNIH LOGIKA

Poglavlje 7

Pascalov svijet

7.1 Vjerojatnost kao logika neizvjesnosti

U ovom poglavlju istražujemo **teoriju vjerojatnosti** kao prirodno proširenje logike sudova, inspirirani Pascalovim pionirskim radom o igrama na sreću i njegovim filozofskim razmatranjima o neizvjesnosti.

> "Srce ima svoje razloge koje razum ne poznaje" (*Le cœur a ses raisons que la raison ne connaît point*) - Blaise Pascal

Pascal je, zajedno s Fermatom, postavio temelje teorije vjerojatnosti 1654. godine, ali njegov doprinos seže dalje - pokazao je kako matematički pristupiti neizvjesnosti i racionalnom odlučivanju.

7.1.1 Od logike sudova do vjerojatnosti

U klasičnoj logici, **elementarni sud** može biti samo istinit ($\top$) ili lažan ($\bot$). Vjerojatnost generalizira ovaj koncept dopuštajući **stupnjeve istinitosti** između 0 i 1:

- Klasična logika: $v(\varphi) \in \{\bot, \top\}$
- Vjerojatnost: $P(\varphi) \in [0, 1]$

gdje $P(\varphi) = 0$ odgovara $\bot$, a $P(\varphi) = 1$ odgovara $\top$.

```python
class Sud:
    """Predstavlja sud s klasičnom ili vjerojatnosnom vrijednošću."""

    def __init__(self, naziv, klasicna_vrijednost=None, vjerojatnost=None):
```

```
        self.naziv = naziv
        self.klasicna = klasicna_vrijednost  # ⊤ ili ⊥
        self.vjerojatnost = vjerojatnost     # [0, 1]

    def __repr__(self):
        if self.vjerojatnost is not None:
            return f"P('{self.naziv}') = {self.vjerojatnost:.2f}"
        elif self.klasicna is not None:
            return f"Sud '{self.naziv}': {'⊤' if self.klasicna else '⊥'}"
        return f"Sud '{self.naziv}': neodređen"

# Primjeri sudova
print("Usporedba logike i vjerojatnosti:")
print("="*34)

# Klasična logika
print("\nKlasična logika:")
kisa_klasicno = Sud("Pada kiša", klasicna_vrijednost=True)
sunce_klasicno = Sud("Sunčano je", klasicna_vrijednost=False)
print(f"  {kisa_klasicno}")
print(f"  {sunce_klasicno}")

# Vjerojatnosna logika
print("\nVjerojatnosna logika:")
kisa_vjerojatnost = Sud("Pada kiša", vjerojatnost=0.3)
sunce_vjerojatnost = Sud("Sunčano je", vjerojatnost=0.7)
oblacno_vjerojatnost = Sud("Oblačno", vjerojatnost=0.45)
print(f"  {kisa_vjerojatnost}")
print(f"  {sunce_vjerojatnost}")
print(f"  {oblacno_vjerojatnost}")

# Interpretacija
print("\nInterpretacija:")
interpretacije = [
    (0.0, "⊥ (sigurno lažno)"),
    (0.3, "stupanj vjerovanja 30%"),
    (0.5, "potpuna neizvjesnost"),
    (0.7, "stupanj vjerovanja 70%"),
    (1.0, "⊤ (sigurno istinito)")
]
for p, opis in interpretacije:
    print(f"  P = {p:.2f} ≈ {opis}")
```

Output

```
Usporedba logike i vjerojatnosti:
==================================

Klasična logika:
  Sud 'Pada kiša': ⊤
  Sud 'Sunčano je': ⊥
```

```
Vjerojatnosna logika:
  P('Pada kiša') = 0.30
  P('Sunčano je') = 0.70
  P('Oblačno') = 0.45

Interpretacija:
  P = 0.00 ≈ ⊥ (sigurno lažno)
  P = 0.30 ≈ stupanj vjerovanja 30%
  P = 0.50 ≈ potpuna neizvjesnost
  P = 0.70 ≈ stupanj vjerovanja 70%
  P = 1.00 ≈ ⊤ (sigurno istinito)
```

7.1.2 Kolmogorovljevi aksiomi vjerojatnosti

Andrej Kolmogorov je 1933. formalizirao teoriju vjerojatnosti kroz tri elegantna aksioma. Za skup svih mogućih sudova Ω i vjerojatnosnu mjeru P:

1. **Nenegativnost**: $P(\varphi) \geq 0$ za svaki sud φ
2. **Normalizacija**: $P(\Omega) = 1$ (sigurna istina ima vjerojatnost 1)
3. **Aditivnost**: Za međusobno isključive sudove $\varphi_1, \varphi_2, ...$:

$$P(\varphi_1 \vee \varphi_2 \vee ...) = P(\varphi_1) + P(\varphi_2) + ...$$

```python
class VjerojatnosniProstor:
    """Predstavlja vjerojatnosni prostor s Kolmogorovljevim aksiomima."""

    def __init__(self, omega, vjerojatnosti):
        """
        omega: skup elementarnih sudova
        vjerojatnosti: rječnik {sud: vjerojatnost}
        """
        self.omega = omega
        self.P = vjerojatnosti
        self._provjeri_aksiome()

    def _provjeri_aksiome(self):
        """Provjerava zadovoljavaju li vjerojatnosti aksiome."""
        # Aksiom 1: Nenegativnost
        for p in self.P.values():
            if p < 0:
                raise ValueError(f"Vjerojatnost {p} krši aksiom nenegativnosti!")

        # Aksiom 2: Normalizacija
        suma = sum(self.P.values())
        if abs(suma - 1.0) > 1e-10:
```

```
            raise ValueError(f"Suma vjerojatnosti {suma} ≠ 1 (krši normalizaciju)!")

    def vjerojatnost_suda(self, predikat):
        """Računa vjerojatnost složenog suda."""
        p = 0
        for ishod, vjerojatnost in self.P.items():
            if predikat(ishod):
                p += vjerojatnost
        return p

# Primjer: Poštena kocka
kocka_omega = {1, 2, 3, 4, 5, 6}
kocka_P = {i: 1/6 for i in kocka_omega}
kocka = VjerojatnosniProstor(kocka_omega, kocka_P)

print("Demonstracija Kolmogorovljevih aksioma")
print("="*39)
print("\nProstor elementarnih sudova Ω:")
print(f"  Kocka pokazuje: {kocka_omega}")

# Aksiom 1: Nenegativnost
print("\nAKSIOM 1 - Nenegativnost:")
testovi = [
    ("Paran broj", lambda x: x % 2 == 0),
    ("Broj > 4", lambda x: x > 4),
    ("Broj = 7", lambda x: x == 7)
]
for naziv, predikat in testovi:
    p = kocka.vjerojatnost_suda(predikat)
    print(f"  P('{naziv}') = {p:.3f} ≥ 0 ✓")

# Aksiom 2: Normalizacija
print("\nAKSIOM 2 - Normalizacija:")
p_omega = kocka.vjerojatnost_suda(lambda x: True)
print(f"  P(Ω) = P('Bilo koji broj 1-6') = {p_omega:.3f} ✓")

# Aksiom 3: Aditivnost
print("\nAKSIOM 3 - Aditivnost:")
print("  Međusobno isključivi sudovi:")
p1 = kocka.vjerojatnost_suda(lambda x: x == 1)
p3 = kocka.vjerojatnost_suda(lambda x: x == 3)
p5 = kocka.vjerojatnost_suda(lambda x: x == 5)
print(f"    P('Broj = 1') = {p1:.3f}")
print(f"    P('Broj = 3') = {p3:.3f}")
print(f"    P('Broj = 5') = {p5:.3f}")

p_unija = kocka.vjerojatnost_suda(lambda x: x in {1, 3, 5})
p_suma = p1 + p3 + p5

print(f"  \n  P('Broj ∈ {{1,3,5}}') = {p_unija:.3f}")
print(f"  P('1') + P('3') + P('5') = {p_suma:.3f}")
print(f"  Aditivnost vrijedi: {p_unija:.3f} = {p_suma:.3f} ✓")

```

```
76 print("\nPosljedice aksioma:")
77 print("  P(¬φ) = 1 - P(φ)")
78 print("  P(∅) = 0")
79 print("  P(φ ∨ ψ) = P(φ) + P(ψ) - P(φ ∧ ψ)")
```

Output

```
Demonstracija Kolmogorovljevih aksioma
========================================

Prostor elementarnih sudova Ω:
  Kocka pokazuje: {1, 2, 3, 4, 5, 6}

AKSIOM 1 - Nenegativnost:
  P('Paran broj') = 0.500 ≥ 0 ✓
  P('Broj > 4') = 0.333 ≥ 0 ✓
  P('Broj = 7') = 0.000 ≥ 0 ✓

AKSIOM 2 - Normalizacija:
  P(Ω) = P('Bilo koji broj 1-6') = 1.000 ✓

AKSIOM 3 - Aditivnost:
  Međusobno isključivi sudovi:
    P('Broj = 1') = 0.167
    P('Broj = 3') = 0.167
    P('Broj = 5') = 0.167

  P('Broj ∈ {1,3,5}') = 0.500
  P('1') + P('3') + P('5') = 0.500
  Aditivnost vrijedi: 0.500 = 0.500 ✓

Posljedice aksioma:
  P(¬φ) = 1 - P(φ)
  P(∅) = 0
  P(φ ∨ ψ) = P(φ) + P(ψ) - P(φ ∧ ψ)
```

7.1.3 Vjerojatnost kao proširenje logičkih veznika

Vjerojatnosni račun generalizira logičke veznike:

- **Negacija**: $P(\neg\varphi) = 1 - P(\varphi)$
- **Disjunkcija**: $P(\varphi \vee \psi) = P(\varphi) + P(\psi) - P(\varphi \wedge \psi)$
- **Konjunkcija**: $P(\varphi \wedge \psi) = P(\varphi) \cdot P(\psi|\varphi)$

Kada su sudovi **nezavisni**: $P(\varphi \wedge \psi) = P(\varphi) \cdot P(\psi)$

```python
class VjerojatnosnaLogika:
    """Implementira vjerojatnosne verzije logičkih veznika."""

    @staticmethod
    def negacija(p_phi):
        """P(¬φ) = 1 - P(φ)"""
        return 1 - p_phi

    @staticmethod
    def disjunkcija(p_phi, p_psi, p_phi_i_psi):
        """P(φ ∨ ψ) = P(φ) + P(ψ) - P(φ ∧ ψ)"""
        return p_phi + p_psi - p_phi_i_psi

    @staticmethod
    def konjunkcija_nezavisna(p_phi, p_psi):
        """P(φ ∧ ψ) = P(φ) · P(ψ) za nezavisne sudove"""
        return p_phi * p_psi

    @staticmethod
    def uvjetna_vjerojatnost(p_phi_i_psi, p_psi):
        """P(φ|ψ) = P(φ ∧ ψ) / P(ψ)"""
        if p_psi == 0:
            return None  # Nedefinirana
        return p_phi_i_psi / p_psi

    @staticmethod
    def implikacija(p_phi, p_psi, p_phi_i_psi):
        """P(φ → ψ) = P(¬φ ∨ ψ)"""
        p_neg_phi = 1 - p_phi
        # P(¬φ ∧ ψ) = P(ψ) - P(φ ∧ ψ)
        p_neg_phi_i_psi = p_psi - p_phi_i_psi
        return p_neg_phi + p_psi - p_neg_phi_i_psi

# Primjer: Vremenske prilike
vl = VjerojatnosnaLogika()

# Definiraj vjerojatnosti
P_kisa = 0.6       # P(pada kiša)
P_hladno = 0.4     # P(hladno je)
P_kisa_i_hladno = 0.3  # P(pada kiša ∧ hladno je)

print("Vjerojatnosni logički veznici")
print("="*30)
print("\nOsnovni sudovi:")
print(f"  P(A) = {P_kisa:.2f}  ('Pada kiša')")
print(f"  P(B) = {P_hladno:.2f}  ('Hladno je')")
print(f"  P(A ∧ B) = {P_kisa_i_hladno:.2f}  ('Pada kiša i hladno je')")

# Negacija
print("\n1. NEGACIJA:")
P_ne_kisa = vl.negacija(P_kisa)
print(f"  P(¬A) = 1 - P(A) = 1 - {P_kisa:.2f} = {P_ne_kisa:.2f}")
```

```
print(f"  Interpretacija: Vjerojatnost da ne pada kiša je {P_ne_kisa*100:.0f}%")

# Disjunkcija
print("\n2. DISJUNKCIJA:")
P_kisa_ili_hladno = vl.disjunkcija(P_kisa, P_hladno, P_kisa_i_hladno)
print(f"  P(A ∨ B) = P(A) + P(B) - P(A ∧ B)")
print(f"  P(A ∨ B) = {P_kisa:.2f} + {P_hladno:.2f} - {P_kisa_i_hladno:.2f} =
↪ {P_kisa_ili_hladno:.2f}")
print(f"  Interpretacija: Vjerojatnost da pada kiša ili je hladno je
↪ {P_kisa_ili_hladno*100:.0f}%")

# Implikacija
print("\n3. IMPLIKACIJA:")
P_ako_kisa_onda_hladno = vl.implikacija(P_kisa, P_hladno, P_kisa_i_hladno)
print(f"  P(A → B) = P(¬A ∨ B) = P(¬A) + P(B) - P(¬A ∧ B)")
P_ne_kisa_i_hladno = P_hladno - P_kisa_i_hladno
print(f"  P(A → B) = {P_ne_kisa:.2f} + {P_hladno:.2f} - {P_ne_kisa_i_hladno:.2f} =
↪ {P_ako_kisa_onda_hladno:.2f}")
print(f"  Interpretacija: Vjerojatnost da 'ako pada kiša, onda je hladno' je
↪ {P_ako_kisa_onda_hladno*100:.0f}%")

# Uvjetna vjerojatnost
print("\n4. UVJETNA VJEROJATNOST:")
P_hladno_ako_kisa = vl.uvjetna_vjerojatnost(P_kisa_i_hladno, P_kisa)
print(f"  P(B|A) = P(A ∧ B) / P(A) = {P_kisa_i_hladno:.2f} / {P_kisa:.2f} =
↪ {P_hladno_ako_kisa:.2f}")
print(f"  Interpretacija: Ako pada kiša, vjerojatnost da je hladno je
↪ {P_hladno_ako_kisa*100:.0f}%")

# Test nezavisnosti
print("\nTest nezavisnosti:")
P_nezavisno = vl.konjunkcija_nezavisna(P_kisa, P_hladno)
print(f"  P(A) · P(B) = {P_kisa:.2f} · {P_hladno:.2f} = {P_nezavisno:.2f}")
print(f"  P(A ∧ B) = {P_kisa_i_hladno:.2f}")
if abs(P_nezavisno - P_kisa_i_hladno) < 0.01:
    print(f"  Budući da {P_nezavisno:.2f} ≈ {P_kisa_i_hladno:.2f}, sudovi su približno
    ↪ nezavisni")
else:
    print(f"  Budući da {P_nezavisno:.2f} ≠ {P_kisa_i_hladno:.2f}, sudovi NISU nezavisni")
    print("  (Kiša i hladnoća su povezani!)")
```

Output

```
Vjerojatnosni logički veznici
==============================

Osnovni sudovi:
  P(A) = 0.60  ('Pada kiša')
  P(B) = 0.40  ('Hladno je')
  P(A ∧ B) = 0.30  ('Pada kiša i hladno je')
```

```
1. NEGACIJA:
  P(¬A) = 1 - P(A) = 1 - 0.60 = 0.40
  Interpretacija: Vjerojatnost da ne pada kiša je 40%

2. DISJUNKCIJA:
  P(A ∨ B) = P(A) + P(B) - P(A ∧ B)
  P(A ∨ B) = 0.60 + 0.40 - 0.30 = 0.70
  Interpretacija: Vjerojatnost da pada kiša
    ili je hladno je 70%

3. IMPLIKACIJA:
  P(A → B) = P(¬A ∨ B) = P(¬A) + P(B) - P(¬A ∧ B)
  P(A → B) = 0.40 + 0.40 - 0.10 = 0.70
  Interpretacija: P('ako pada kiša, onda je hladno') = 70%

4. UVJETNA VJEROJATNOST:
  P(B|A) = P(A ∧ B) / P(A) = 0.30 / 0.60 = 0.50
  Interpretacija: Ako pada kiša, vjerojatnost da je hladno je 50%

Test nezavisnosti:
  P(A) · P(B) = 0.60 · 0.40 = 0.24
  P(A ∧ B) = 0.30
  Budući da 0.24 ≠ 0.30, sudovi NISU nezavisni
  (Kiša i hladnoća su povezani!)
```

7.1.4 Bayesov teorem - zaključivanje s neizvjesnošću

Bayesov teorem omogućava ažuriranje vjerovanja na temelju novih dokaza:

> "Vjerojatnost je vodič života" - Ciceron, anticipirajući Bayesovu logiku

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

gdje je:

- $P(H)$ - **apriorna vjerojatnost** hipoteze
- $P(E|H)$ - **izglednost** dokaza ako je hipoteza istinita
- $P(H|E)$ - **aposteriorna vjerojatnost** nakon dokaza

```python
class BayesovoZakljucivanje:
    """Implementira Bayesovo ažuriranje vjerojatnosti."""

```

```
    def __init__(self, apriorna, izglednosti):
        """
        apriorna: dict {hipoteza: P(hipoteza)}
        izglednosti: dict {hipoteza: {dokaz: P(dokaz|hipoteza)}}
        """
        self.apriorna = apriorna
        self.izglednosti = izglednosti

    def azuriraj(self, dokaz):
        """Ažurira vjerojatnosti na temelju novog dokaza."""
        # Izračunaj P(dokaz)
        p_dokaz = 0
        for hipoteza, p_h in self.apriorna.items():
            p_dokaz += self.izglednosti[hipoteza][dokaz] * p_h

        # Bayesovo ažuriranje za svaku hipotezu
        aposteriorna = {}
        for hipoteza, p_h in self.apriorna.items():
            p_dokaz_ako_h = self.izglednosti[hipoteza][dokaz]
            aposteriorna[hipoteza] = (p_dokaz_ako_h * p_h) / p_dokaz

        return aposteriorna, p_dokaz

# Primjer: Medicinska dijagnoza
print("Bayesovo zaključivanje - Medicinska dijagnoza")
print("="*46)
print("\nScenarij: Test za rijetku bolest\n")

# Definiraj vjerojatnosti
apriorna = {
    "bolest": 0.001,     # 1 od 1000 ljudi ima bolest
    "zdrav": 0.999
}

izglednosti = {
    "bolest": {
        "pozitivan": 0.99,  # Osjetljivost testa (true positive rate)
        "negativan": 0.01   # False negative rate
    },
    "zdrav": {
        "pozitivan": 0.05,  # False positive rate
        "negativan": 0.95   # True negative rate
    }
}

bayes = BayesovoZakljucivanje(apriorna, izglednosti)

print("Početne informacije:")
print(f"  P(Bolest) = {apriorna['bolest']:.3f} (1 od 1000 ljudi ima bolest)")
print(f"  P(Test+|Bolest) = {izglednosti['bolest']['pozitivan']:.3f} (osjetljivost testa)")
print(f"  P(Test+|¬Bolest) = {izglednosti['zdrav']['pozitivan']:.3f} (lažno pozitivna
↪  stopa)")

```

```
print("\nPitanje: Ako je test pozitivan, kolika je vjerojatnost bolesti?")

# Bayesovo ažuriranje
aposteriorna, p_pozitivan = bayes.azuriraj("pozitivan")

print("\nBayesov račun:")
print("1. P(Test+) = P(Test+|Bolest)·P(Bolest) + P(Test+|¬Bolest)·P(¬Bolest)")
p1 = izglednosti['bolest']['pozitivan'] * apriorna['bolest']
p2 = izglednosti['zdrav']['pozitivan'] * apriorna['zdrav']
print(f"   P(Test+) = {izglednosti['bolest']['pozitivan']:.3f} · {apriorna['bolest']:.3f} + "
      f"{izglednosti['zdrav']['pozitivan']:.3f} · {apriorna['zdrav']:.3f}")
print(f"   P(Test+) = {p1:.5f} + {p2:.5f} = {p_pozitivan:.5f}")

print("\n2. P(Bolest|Test+) = P(Test+|Bolest) · P(Bolest) / P(Test+)")
print(f"   P(Bolest|Test+) = {izglednosti['bolest']['pozitivan']:.3f} · "
      f"{apriorna['bolest']:.3f} / {p_pozitivan:.5f}")
print(f"   P(Bolest|Test+) = {aposteriorna['bolest']:.5f}")

print("\nZAKLJUČAK:")
print(f"  Apriorna vjerojatnost: {apriorna['bolest']*100:.1f}%")
print(f"  Aposteriorna vjerojatnost (nakon pozitivnog testa):
↪ {aposteriorna['bolest']*100:.1f}%")
print("  \n  Iako je test vrlo pouzdan (99% osjetljivost),")
print(f"  pozitivan test povećava vjerojatnost bolesti samo na
↪ {aposteriorna['bolest']*100:.1f}%!")
print("  \n  Razlog: Bolest je vrlo rijetka, pa većina pozitivnih")
print("  testova su lažno pozitivni.")

# Vizualizacija
print("\nVizualizacija od 100,000 ljudi:")
n_ljudi = 100000
n_bolesnih = int(n_ljudi * apriorna['bolest'])
n_zdravih = n_ljudi - n_bolesnih

tp = int(n_bolesnih * izglednosti['bolest']['pozitivan'])  # True positive
fn = n_bolesnih - tp  # False negative
fp = int(n_zdravih * izglednosti['zdrav']['pozitivan'])  # False positive
tn = n_zdravih - fp  # True negative

print(f"  Bolesni: {n_bolesnih} ljudi")
print(f"    → Pozitivan test: {tp} (istinito pozitivni)")
print(f"    → Negativan test: {fn} (lažno negativan)")
print(f"  \n  Zdravi: {n_zdravih:,} ljudi")
print(f"    → Pozitivan test: {fp:,} (lažno pozitivni)")
print(f"    → Negativan test: {tn:,} (istinito negativni)")
print(f"  \n  Ukupno pozitivnih testova: {tp + fp:,}")
print(f"  Od toga stvarno bolesnih: {tp}")
print(f"  Vjerojatnost bolesti kod pozitivnog testa: {tp}/{tp+fp} = {tp/(tp+fp)*100:.1f}%")
```

Output

```
Bayesovo zaključivanje - Medicinska dijagnoza
==============================================

Scenarij: Test za rijetku bolest

Početne informacije:
  P(Bolest) = 0.001 (1 od 1000 ljudi ima bolest)
  P(Test+|Bolest) = 0.990 (osjetljivost testa)
  P(Test+|¬Bolest) = 0.050 (lažno pozitivna stopa)

Pitanje: Ako je test pozitivan, kolika je vjerojatnost bolesti?

Bayesov račun:
1. P(Test+) = P(Test+|Bolest)·P(Bolest) + P(Test+|¬Bolest)·P(¬Bolest)
   P(Test+) = 0.990 · 0.001 + 0.050 · 0.999
   P(Test+) = 0.00099 + 0.04995 = 0.05094

2. P(Bolest|Test+) = P(Test+|Bolest) · P(Bolest) / P(Test+)
   P(Bolest|Test+) = 0.990 · 0.001 / 0.05094
   P(Bolest|Test+) = 0.01943

ZAKLJUČAK:
  Apriorna vjerojatnost: 0.1%
  Aposteriorna vjerojatnost (nakon pozitivnog testa): 1.9%

  Iako je test vrlo pouzdan (99% osjetljivost),
  pozitivan test povećava vjerojatnost bolesti samo na 1.9%!

  Razlog: Bolest je vrlo rijetka, pa većina pozitivnih
  testova su lažno pozitivni.

Vizualizacija od 100,000 ljudi:
  Bolesni: 100 ljudi
    → Pozitivan test: 99 (istinito pozitivni)
    → Negativan test: 1 (lažno negativan)

  Zdravi: 99,900 ljudi
    → Pozitivan test: 4,995 (lažno pozitivni)
    → Negativan test: 94,905 (istinito negativni)

  Ukupno pozitivnih testova: 5,094
  Od toga stvarno bolesnih: 99
  Vjerojatnost bolesti kod pozitivnog testa: 99/5094 = 1.9%
```

7.1.5 Pascalova oklada - filozofija vjerojatnosti

Pascal je primijenio vjerojatnosno razmišljanje na ultimativno filozofsko pitanje - postojanje Boga:

"Morate se kladiti. To nije dobrovoljno, primorani ste na to" - Pascal, *Pensées*

Pascalova matrica odlučivanja:

| | Bog postoji | Bog ne postoji | |—|—————|—————| | **Vjerujem** | $+\infty$ (vječna sreća) | $-c$ (mali gubitak) | | **Ne vjerujem** | $-\infty$ (vječna kazna) | $+c$ (mali dobitak) |

Čak i ako je $P(\text{Bog postoji}) = \epsilon$ vrlo mala, očekivana korisnost vjerovanja je beskonačna!

```python
import math

def fmt_prob(x: float) -> str:
    return f"{x:.2f}" if x >= 0.01 else f"{x:.4f}"

class PascalovaOklada:
    """Simulacija Pascalove oklade kroz teoriju odlučivanja."""

    def __init__(self):
        # Matrica korisnosti (payoff matrix)
        # Koristimo velike brojeve umjesto beskonačnosti za demonstraciju
        self.korisnost = {
            ("vjeruj", "postoji"): float('inf'),    # Vječna sreća
            ("vjeruj", "ne_postoji"): -10,          # Mali gubitak (trud vjerovanja)
            ("ne_vjeruj", "postoji"): float('-inf'), # Vječna kazna
            ("ne_vjeruj", "ne_postoji"): 10         # Mali dobitak (sloboda)
        }

    def ocekivana_korisnost(self, akcija, p_bog):
        """Računa očekivanu korisnost akcije."""
        k_postoji = self.korisnost[(akcija, "postoji")]
        k_ne_postoji = self.korisnost[(akcija, "ne_postoji")]

        return p_bog * k_postoji + (1 - p_bog) * k_ne_postoji

    def optimalna_odluka(self, p_bog):
        """Vraća optimalnu odluku za danu vjerojatnost."""
        eu_vjeruj = self.ocekivana_korisnost("vjeruj", p_bog)
        eu_ne_vjeruj = self.ocekivana_korisnost("ne_vjeruj", p_bog)

        if eu_vjeruj > eu_ne_vjeruj:
            return "vjeruj", eu_vjeruj, eu_ne_vjeruj
        elif eu_ne_vjeruj > eu_vjeruj:
            return "ne_vjeruj", eu_vjeruj, eu_ne_vjeruj
        else:
            return "svejedno", eu_vjeruj, eu_ne_vjeruj

# Demonstracija
oklada = PascalovaOklada()

print("Pascalova oklada - Analiza odlučivanja")
```

```
print("="*39)
print("\nMatrica korisnosti:")
print("                 Bog postoji    Bog ne postoji")
print("  Vjerujem:      +∞             -10           ")
print("  Ne vjerujem:   -∞             +10           ")

print("\nAnaliza za različite vjerojatnosti postojanja Boga:\n")

vjerojatnosti = [0.5, 0.1, 0.01, 0.0001]
for p in vjerojatnosti:
    odluka, eu_v, eu_nv = oklada.optimalna_odluka(p)

    # Example usage (replace your print block with this structure)
    eu_v_str = f"{eu_v:.2f}" if (eu_v not in (float('inf'), float('-inf'))) else ("∞" if
    ↪  eu_v > 0 else "-∞")
    eu_nv_str = f"{eu_nv:.2f}" if (eu_nv not in (float('inf'), float('-inf'))) else ("∞" if
    ↪  eu_nv > 0 else "-∞")

    p_str = fmt_prob(p)
    one_minus_p_str = fmt_prob(1 - p)

    print(f"  E[vjerujem] = {p_str} · ∞ + {one_minus_p_str} · (-10) = {eu_v_str}")
    print(f"  E[ne vjerujem] = {p_str} · (-∞) + {one_minus_p_str} · 10 = {eu_nv_str}")
    print(f"  → Racionalna odluka: {'VJERUJ' if odluka == 'vjeruj' else 'NE VJERUJ'}\n")

print("Pascalov zaključak:")
print("  Čak i uz infinitezimalnu vjerojatnost postojanja Boga,")
print("  racionalno je vjerovati zbog beskonačne nagrade/kazne.")

print("\nFilozofske kritike:")
kritike = [
    "Problem mnoštva religija (koja vjera?)",
    "Autentičnost vjerovanja (može li se 'odlučiti' vjerovati?)",
    "Beskonačnost kao korisnost (ima li smisla?)",
    "Moralni prigovor (je li to pravo vjerovanje?)"
]
for i, kritika in enumerate(kritike, 1):
    print(f"  {i}. {kritika}")

# Konačna verzija
print("\nKonačna aproksimacija (bez beskonačnosti):")
print("  Korisnosti: vjeruj(Bog da)=1000, vjeruj(Bog ne)=-10")
print("              ne vjeruj(Bog da)=-1000, ne vjeruj(Bog ne)=10")

# Kritična vjerojatnost
# 1000p - 10(1-p) = -1000p + 10(1-p)
# 1000p - 10 + 10p = -1000p + 10 - 10p
# 2020p = 20
p_kritična = 20 / 2020
print(f"  \n  Kritična vjerojatnost: P* = {p_kritična:.3f}")
print(f"  Ako P(Bog) > {p_kritična:.3f}, vjeruj; inače ne vjeruj.")
```

Output

```
Pascalova oklada - Analiza odlučivanja
======================================

Matrica korisnosti:
                  Bog postoji    Bog ne postoji
  Vjerujem:       +∞              -10
  Ne vjerujem:    -∞              +10

Analiza za različite vjerojatnosti postojanja Boga:

  E[vjerujem] = 0.50 · ∞ + 0.50 · (-10) = ∞
  E[ne vjerujem] = 0.50 · (-∞) + 0.50 · 10 = -∞
  → Racionalna odluka: VJERUJ

  E[vjerujem] = 0.10 · ∞ + 0.90 · (-10) = ∞
  E[ne vjerujem] = 0.10 · (-∞) + 0.90 · 10 = -∞
  → Racionalna odluka: VJERUJ

  E[vjerujem] = 0.01 · ∞ + 0.99 · (-10) = ∞
  E[ne vjerujem] = 0.01 · (-∞) + 0.99 · 10 = -∞
  → Racionalna odluka: VJERUJ

  E[vjerujem] = 0.0001 · ∞ + 1.00 · (-10) = ∞
  E[ne vjerujem] = 0.0001 · (-∞) + 1.00 · 10 = -∞
  → Racionalna odluka: VJERUJ

Pascalov zaključak:
  Čak i uz infinitezimalnu vjerojatnost postojanja Boga,
  racionalno je vjerovati zbog beskonačne nagrade/kazne.

Filozofske kritike:
  1. Problem mnoštva religija (koja vjera?)
  2. Autentičnost vjerovanja (može li se 'odlučiti' vjerovati?)
  3. Beskonačnost kao korisnost (ima li smisla?)
  4. Moralni prigovor (je li to pravo vjerovanje?)

Konačna aproksimacija (bez beskonačnosti):
  Korisnosti: vjeruj(Bog da)=1000, vjeruj(Bog ne)=-10
              ne vjeruj(Bog da)=-1000, ne vjeruj(Bog ne)=10

  Kritična vjerojatnost: P* = 0.010
  Ako P(Bog) > 0.010, vjeruj; inače ne vjeruj.
```

7.1.6 Problem indukcije i vjerojatnost

David Hume je ukazao na **problem indukcije** - nemožnost logičkog opravdanja generalizacije iz konačnog broja opažanja. Vjerojatnost nudi djelomično rješenje:

"Navika je veliki vodič ljudskog života" - Hume

Umjesto traženja sigurnosti, vjerojatnost kvantificira stupanj racionalnog vjerovanja.

```python
class InduktivnoZakljucivanje:
    """Modelira induktivno zaključivanje kroz vjerojatnost."""

    def __init__(self):
        self.povijest = []

    def laplace_sukcesija(self, uspjesi, pokusaji):
        """Laplace-ov zakon sukcesije.

        P(sljedeći uspjeh) = (k + 1) / (n + 2)
        gdje je k broj uspjeha, n broj pokušaja
        """
        return (uspjesi + 1) / (pokusaji + 2)

    def azuriraj_vjerovanje(self, novi_dokaz):
        """Ažurira vjerovanje na temelju novog dokaza."""
        self.povijest.append(novi_dokaz)
        uspjesi = sum(self.povijest)
        pokusaji = len(self.povijest)
        return self.laplace_sukcesija(uspjesi, pokusaji)

# Primjer: Sunce izlazi svaki dan
print("Problem indukcije - Sunce izlazi")
print("="*33)

indukcija = InduktivnoZakljucivanje()

print("\nLaplace-ov zakon sukcesije:")
print("  P(uspjeh) = (k + 1) / (n + 2)")
print("  gdje je k = broj dosadašnjih uspjeha, n = ukupan broj pokušaja")

print("\nDana\tP(sunce izađe)\tPromjena")
print("-"*4 + "\t" + "-"*14 + "\t" + "-"*8)

# Simuliraj promatranje sunca
dani = [0, 1, 2, 3, 4, 5, 10, 30, 100, 365, 1000, 10000]
prethodna_p = 0.5

for dan in dani:
    p = indukcija.laplace_sukcesija(dan, dan)
    promjena = p - prethodna_p if dan > 0 else 0

    print(f"{dan}\t{p:.4f}\t\t", end="")
    if dan > 0:
        print(f"+{promjena:.4f}" if promjena > 0 else f"{promjena:.4f}")
    else:
        print("-")
```

```
48
49      if dan in [1, 5, 30, 365, 10000]:
50          prethodna_p = p
51
52  print("\nHumeov problem:")
53  print("  Koliko god puta sunce izašlo, P(sunce izađe sutra) < 1")
54  print("  Nikad ne možemo biti potpuno sigurni!")
55
56  # Crni labud
57  print("\nCrni labud scenarij:")
58  bijeli_labudovi = 1000
59  p_bijeli = indukcija.laplace_sukcesija(bijeli_labudovi, bijeli_labudovi)
60  print(f"Nakon {bijeli_labudovi} bijelih labudova, P(sljedeći bijel) = {p_bijeli:.4f}")
61  print("Ali jedan crni labud mijenja sve!")
62
63  # Nakon crnog labuda
64  ukupno = bijeli_labudovi + 1
65  p_bijeli_novi = indukcija.laplace_sukcesija(bijeli_labudovi, ukupno)
66  p_crni = indukcija.laplace_sukcesija(1, ukupno)
67
68  print("\nNakon crnog labuda:")
69  print(f"  Vidjeli: {bijeli_labudovi} bijelih, 1 crni")
70  print(f"  P(sljedeći bijel) = {p_bijeli_novi:.4f}")
71  print(f"  P(sljedeći crni) = {p_crni:.4f}")
72
73  print("\nFilozofska pouka:")
74  print("  Indukcija ne daje sigurnost, već racionalne stupnjeve vjerovanja.")
75  print("  Vjerojatnost kvantificira našu neizvjesnost o budućnosti.")
```

Output

```
Problem indukcije - Sunce izlazi
================================

Laplace-ov zakon sukcesije:
  P(uspjeh) = (k + 1) / (n + 2)
  gdje je k = broj dosadašnjih uspjeha, n = ukupan broj pokušaja

Dana P(sunce izađe) Promjena
---- -------------- --------
0 0.5000  -
1 0.6667  +0.1667
2 0.7500  +0.0833
3 0.8000  +0.1333
4 0.8333  +0.1667
5 0.8571  +0.1905
10 0.9167  +0.0595
30 0.9688  +0.1116
100 0.9902  +0.0214
365 0.9973  +0.0285
1000 0.9990  +0.0017
```

```
10000 0.9999  +0.0026

Humeov problem:
  Koliko god puta sunce izašlo, P(sunce izađe sutra) < 1
  Nikad ne možemo biti potpuno sigurni!

Crni labud scenarij:
Nakon 1000 bijelih labudova, P(sljedeći bijel) = 0.9990
Ali jedan crni labud mijenja sve!

Nakon crnog labuda:
  Vidjeli: 1000 bijelih, 1 crni
  P(sljedeći bijel) = 0.9980
  P(sljedeći crni) = 0.0020

Filozofska pouka:
  Indukcija ne daje sigurnost, već racionalne stupnjeve vjerovanja.
  Vjerojatnost kvantificira našu neizvjesnost o budućnosti.
```

7.1.7 Monty Hall problem - paradoksi uvjetne vjerojatnosti

Monty Hall problem ilustrira kako naša intuicija često griješi kod uvjetnih vjerojatnosti:

```python
import random

class MontyHall:
    """Simulacija Monty Hall problema."""

    def igraj(self, strategija="promijeni"):
        """Igra jednu rundu Monty Hall igre.

        strategija: 'ostani' ili 'promijeni'
        """
        # Postavi igru
        vrata = [1, 2, 3]
        auto = random.choice(vrata)

        # Igrač bira
        izbor = random.choice(vrata)

        # Voditelj otvara vrata s kozom
        moguca_vrata = [v for v in vrata if v != izbor and v != auto]
        voditelj_otvara = random.choice(moguca_vrata) if moguca_vrata else \
                          [v for v in vrata if v != izbor][0]

        # Primijeni strategiju
        if strategija == "promijeni":
            preostala = [v for v in vrata if v != izbor and v != voditelj_otvara]
```

```
            konacni_izbor = preostala[0]
        else:  # ostani
            konacni_izbor = izbor

        return konacni_izbor == auto

    def simuliraj(self, n_igara=10000):
        """Simulira više igara s obje strategije."""
        rezultati = {
            "ostani": 0,
            "promijeni": 0
        }

        for _ in range(n_igara):
            if self.igraj("ostani"):
                rezultati["ostani"] += 1
            if self.igraj("promijeni"):
                rezultati["promijeni"] += 1

        return rezultati, n_igara

# Simulacija
mh = MontyHall()
rezultati, n = mh.simuliraj(10000)

print("Monty Hall Problem - Simulacija")
print("="*32)
print("\nPostavka:")
print("  3 vrata: iza jednih je automobil, iza drugih dvoje koze")
print("  1. Birate vrata")
print("  2. Voditelj otvara druga vrata s kozom")
print("  3. Možete promijeniti izbor ili ostati")

print(f"\nSimulacija {n:,} igara:\n")

for strategija, pobjede in rezultati.items():
    postotak = (pobjede / n) * 100
    print(f"Strategija {strategija.upper()}:")
    print(f"  Pobjede: {pobjede:,} / {n:,}")
    print(f"  Postotak pobjede: {postotak:.1f}%\n")

# Bayesova analiza
print("Bayesova analiza:")
print("="*18)
print("\nPočetne vjerojatnosti:")
print("  P(auto na vratima 1) = 1/3")
print("  P(auto na vratima 2) = 1/3")
print("  P(auto na vratima 3) = 1/3")

print("\nBirate vrata 1. Voditelj otvara vrata 3 (koza).")

print("\nAžurirane vjerojatnosti:")
print("  P(auto na 1 | voditelj otvorio 3) = 1/3")
```

```
79 print("  P(auto na 2 | voditelj otvorio 3) = 2/3")
80 print("  P(auto na 3 | voditelj otvorio 3) = 0")
81
82 print("\nObjašnjenje:")
83 print("  Voditelj ZNA gdje je auto i MORA otvoriti kozu.")
84 print("  Njegovo otvaranje daje informaciju!")
85 print("  ")
86 print("  Ako je auto na vratima 1: voditelj bira između 2 i 3 (P=1/2)")
87 print("  Ako je auto na vratima 2: voditelj MORA otvoriti 3 (P=1)")
88 print("  Ako je auto na vratima 3: voditelj MORA otvoriti 2 (P=1)")
89 print("  ")
90 print("  Činjenica da je otvorio 3 favorizira vrata 2!")
91
92 print("\nZaključak: UVIJEK se isplati promijeniti izbor!")
```

Output

```
Monty Hall Problem - Simulacija
================================

Postavka:
  3 vrata: iza jednih je automobil, iza drugih dvoje koze
  1. Birate vrata
  2. Voditelj otvara druga vrata s kozom
  3. Možete promijeniti izbor ili ostati

Simulacija 10,000 igara:

Strategija OSTANI:
  Pobjede: 3,328 / 10,000
  Postotak pobjede: 33.3%

Strategija PROMIJENI:
  Pobjede: 6,718 / 10,000
  Postotak pobjede: 67.2%

Bayesova analiza:
==================

Početne vjerojatnosti:
  P(auto na vratima 1) = 1/3
  P(auto na vratima 2) = 1/3
  P(auto na vratima 3) = 1/3

Birate vrata 1. Voditelj otvara vrata 3 (koza).

Ažurirane vjerojatnosti:
  P(auto na 1 | voditelj otvorio 3) = 1/3
  P(auto na 2 | voditelj otvorio 3) = 2/3
  P(auto na 3 | voditelj otvorio 3) = 0
```

```
Objašnjenje:
  Voditelj ZNA gdje je auto i MORA otvoriti kozu.
  Njegovo otvaranje daje informaciju!

  Ako je auto na vratima 1: voditelj bira između 2 i 3 (P=1/2)
  Ako je auto na vratima 2: voditelj MORA otvoriti 3 (P=1)
  Ako je auto na vratima 3: voditelj MORA otvoriti 2 (P=1)

  Činjenica da je otvorio 3 favorizira vrata 2!

Zaključak: UVIJEK se isplati promijeniti izbor!
```

7.1.8 Filozofske interpretacije vjerojatnosti

Postoje različite filozofske interpretacije što vjerojatnost zapravo znači:

1. **Klasična** (Laplace): Omjer povoljnih i mogućih ishoda
2. **Frekventistička** (von Mises): Granična relativna frekvencija
3. **Propenzitetna** (Popper): Objektivna tendencija
4. **Subjektivna/Bayesova** (de Finetti): Stupanj racionalnog vjerovanja

Svaka interpretacija ima filozofske implikacije za prirodu slučajnosti, determinizma i znanja.

```python
def filozofske_interpretacije():
    """Demonstrira različite filozofske interpretacije vjerojatnosti."""

    print("Filozofske interpretacije vjerojatnosti")
    print("="*40)

    # 1. Klasična
    print("\n1. KLASIČNA INTERPRETACIJA (Laplace)")
    print("   'Vjerojatnost je omjer povoljnih i jednakovjerojatnih ishoda'")
    print("   \n   Primjer: Kocka")
    povoljni = 3  # parni brojevi: 2, 4, 6
    mogući = 6
    p_klasična = povoljni / mogući
    print(f"   P(paran broj) = {povoljni}/{mogući} = {p_klasična:.3f}")
    print("   \n   Problem: Pretpostavlja jednakovjerojatnost (cirkularnost)")

    # 2. Frekventistička
    print("\n2. FREKVENTISTIČKA INTERPRETACIJA (von Mises)")
    print("   'Vjerojatnost je granična relativna frekvencija'")
    print("   ")

```

```
    random.seed(42)
    for n in [100, 1000, 10000, 100000]:
        parni = sum(1 for _ in range(n) if random.randint(1, 6) % 2 == 0)
        frekvencija = parni / n
        print(f"   Bacanja: {n}, Parni: {parni}, Frekvencija: {frekvencija:.3f}")
    print("   → Konvergira prema 0.500")
    print("   \n   Problem: Što s jedinstvenim događajima?")

    # 3. Propenzitetna
    print("\n3. PROPENZITETNA INTERPRETACIJA (Popper)")
    print("   'Vjerojatnost je objektivna tendencija sustava'")
    print("   ")
    print("   Kocka ima propenzitet 0.5 za paran broj")
    print("   zbog svoje fizičke strukture.")
    print("   \n   Problem: Kako mjeriti propenzitet?")

    # 4. Subjektivna
    print("\n4. SUBJEKTIVNA INTERPRETACIJA (de Finetti)")
    print("   'Vjerojatnost je stupanj racionalnog vjerovanja'")
    print("   ")
    print("   Moje vjerovanje:")
    print("   P(kiša sutra) = 0.30 (na temelju prognoze)")
    print("   ")
    print("   Koherentnost: Moja vjerovanja moraju zadovoljiti")
    print("   aksiome vjerojatnosti inače mogu biti 'Dutch booked'")
    print("   \n   Problem: Subjektivnost vs. objektivnost")

    # Filozofske implikacije
    print("\nFilozofske implikacije:")
    print("="*24)

    print("\nDETERMINIZAM vs. INDETERMINIZAM:")
    print("  - Je li vjerojatnost samo naša neznanja (epistemička)?")
    print("  - Ili je svijet inherentno slučajan (ontološka)?")

    print("\nPROBLEM REFERENTNE KLASE:")
    print("  - Jesam li '35-godišnjak', 'Hrvat', 'filozof'?")
    print("  - Različite klase daju različite vjerojatnosti!")

    print("\nPRINCIP INDIFERENCIJE:")
    print("  - Bez informacija, dodjeli jednake vjerojatnosti?")
    print("  - Bertrandov paradoks pokazuje probleme")

    print("\n> \"Bog ne igra kocke\" - Einstein")
    print("> \"Einstein, prestani govoriti Bogu što da radi\" - Bohr")

filozofske_interpretacije()
```

Output

```
Filozofske interpretacije vjerojatnosti
=======================================

1. KLASIČNA INTERPRETACIJA (Laplace)
   'Vjerojatnost je omjer povoljnih i jednakovjerojatnih ishoda'

   Primjer: Kocka
   P(paran broj) = 3/6 = 0.500

   Problem: Pretpostavlja jednakovjerojatnost (cirkularnost)

2. FREKVENTISTIČKA INTERPRETACIJA (von Mises)
   'Vjerojatnost je granična relativna frekvencija'

   Bacanja: 100, Parni: 49, Frekvencija: 0.490
   Bacanja: 1000, Parni: 510, Frekvencija: 0.510
   Bacanja: 10000, Parni: 5057, Frekvencija: 0.506
   Bacanja: 100000, Parni: 50215, Frekvencija: 0.502
   → Konvergira prema 0.500

   Problem: Što s jedinstvenim događajima?

3. PROPENZITETNA INTERPRETACIJA (Popper)
   'Vjerojatnost je objektivna tendencija sustava'

   Kocka ima propenzitet 0.5 za paran broj
   zbog svoje fizičke strukture.

   Problem: Kako mjeriti propenzitet?

4. SUBJEKTIVNA INTERPRETACIJA (de Finetti)
   'Vjerojatnost je stupanj racionalnog vjerovanja'

   Moje vjerovanje:
   P(kiša sutra) = 0.30 (na temelju prognoze)

   Koherentnost: Moja vjerovanja moraju zadovoljiti
   aksiome vjerojatnosti inače mogu biti 'Dutch booked'

   Problem: Subjektivnost vs. objektivnost

Filozofske implikacije:
=======================

DETERMINIZAM vs. INDETERMINIZAM:
  - Je li vjerojatnost samo naša neznanja (epistemička)?
  - Ili je svijet inherentno slučajan (ontološka)?

PROBLEM REFERENTNE KLASE:
```

```
  - Jesam li '35-godišnjak', 'Hrvat', 'filozof'?
  - Različite klase daju različite vjerojatnosti!

PRINCIP INDIFERENCIJE:
  - Bez informacija, dodjeli jednake vjerojatnosti?
  - Bertrandov paradoks pokazuje probleme

> "Bog ne igra kocke" - Einstein
> "Einstein, prestani govoriti Bogu što da radi" - Bohr
```

7.1.9 Prijedlozi za daljnje istraživanje

Za produbljivanje razumijevanja vjerojatnosti kao proširenja logike kroz praktične Python zadatke:

Fuzzy logika

Implementirajte fuzzy logičke operatore gdje istinite vrijednosti mogu biti bilo koji broj između 0 i 1. Definirajte Łukasiewicz, Gödel i product t-norme. Pokažite kako se klasična logika dobiva kao granični slučaj.

Cox-Jaynesovi teoremi

Dokažite da svaki sustav koji zadovoljava razumne uvjete za stupnjeve vjerovanja mora biti izomorfan s teorijom vjerojatnosti. Implementirajte alternativni sustav i pokažite kako se svodi na vjerojatnost.

Dutch book argument

Simulirajte klađenje gdje agent s nekoherentnim vjerovanjima (koja krše aksiome vjerojatnosti) uvijek gubi novac. Vizualizirajte kako koherentnost štiti od sigurnog gubitka.

Dempster-Shafer teorija

Implementirajte teoriju dokaza koja generalizira vjerojatnost dopuštajući "ne znam" odgovore. Pokažite kombiniranje dokaza i rad s intervalima vjerojatnosti.

Paradoks spavajuće ljepotice

Simulirajte ovaj paradoks samolociranja u vremenu. Analizirajte sukob između "halfer" i "thirder" pozicija. Povežite s antropijskim principom.

Kvantna vjerojatnost

Implementirajte Born rule i pokažite kako kvantne amplitude daju vjerojatnosti. Demonstrirajte narušavanje Bell-ovih nejednakosti klasičnom vjerojatnošću.

Algoritamska vjerojatnost

Implementirajte Solomonoff-ovu indukciju koristeći Kolmogorovljevu složenost. Pokažite kako kraći programi imaju veću apriornu vjerojatnost.

Kauzalno zaključivanje

Implementirajte Pearl-ove kauzalne grafove. Pokažite razliku između korelacije i kauzalnosti. Simulirajte Simpson-ov paradoks.

Maksimalna entropija

Implementirajte princip maksimalne entropije za izbor vjerojatnosnih distribucija. Pokažite kako ograničenja određuju optimalnu distribuciju.

Vjerojatnosno programiranje

Stvorite jednostavan vjerojatnosni programski jezik gdje varijable mogu biti distribucije. Implementirajte inferenciju kroz MCMC ili variational Bayes.

Svaki zadatak postupno gradi razumijevanje vjerojatnosti kao generalizacije logike, omogućavajući studentima da istraže dublje veze između logike, vjerojatnosti i racionalnog zaključivanja.

7.1.10 Zaključak

Kroz ovu implementaciju istražili smo teoriju vjerojatnosti kao prirodno proširenje logike sudova:

1. **Generalizacija istinitosti** - od $\bot, \top$ do $[0, 1]$
2. **Kolmogorovljevi aksiomi** kao temelj matematičke teorije
3. **Bayesovo zaključivanje** za ažuriranje vjerovanja
4. **Filozofske implikacije** za racionalnost i odlučivanje

Pascal je anticipirao modernu teoriju odlučivanja pokazujući kako matematički pristupiti neizvjesnosti:

> "Razum nas ne može odlučiti između kršćanstva i ateizma, ali morate birati kako ćete živjeti" - Pascal

Vjerojatnost premošćuje jaz između čiste logike i nesigurnog svijeta iskustva. Ona ne daje apsolutnu istinu, ali omogućava racionalno navigiranje kroz neizvjesnost.

Kroz praktično programiranje otkrivamo da je vjerojatnost više od matematičkog alata - ona je jezik za izražavanje stupnjeva vjerovanja, kvantificiranje neizvjesnosti i donošenje racionalnih odluka u svijetu gdje je potpuno znanje nedostižno.

> "Život je umijeće izvlačenja dovoljnih zaključaka iz nedovoljnih premisa" - Samuel Butler

Teorija vjerojatnosti je upravo to umijeće, formalizirano u elegantnom matematičkom okviru.

Poglavlje 8

Bayesov svijet

8.1 Uvjetna vjerojatnost i priroda znanja

U ovom poglavlju istražujemo **Bayesov teorem** - možda najvažniju formulu u epistemologiji, koja mijenja naše razumijevanje znanja, znanosti i racionalnog vjerovanja.

> "Kada činjenice mijenjaju moje mišljenje, ja ga promijenim. Što Vi činite, gospodine?" - John Maynard Keynes

Thomas Bayes (1701-1761), prezbiterijanski svećenik i matematičar, ostavio nam je revolucionarni pristup razumijevanju kako ažurirati vjerovanja na temelju dokaza. Njegov rad, objavljen posthumno 1763., postavlja temelje za razumijevanje znanja kao dinamičkog procesa.

8.1.1 Uvjetna vjerojatnost - temelj Bayesovog razmišljanja

Uvjetna vjerojatnost $P(A|B)$ predstavlja vjerojatnost suda A uz pretpostavku da je sud B istinit.

$$P(A|B) = \frac{P(A \wedge B)}{P(B)}$$

Ova formula kodira fundamentalnu epistemološku ideju: **znanje mijenja vjerojatnosti**.

> "Svo znanje degenerira u vjerojatnost" - David Hume

```
class UvjetnaVjerojatnost:
    """Implementira uvjetnu vjerojatnost s epistemološkom interpretacijom."""

```

```
    def __init__(self, naziv=""):
        self.naziv = naziv
        self.vjerojatnosti = {}

    def postavi(self, sud, vjerojatnost):
        """Postavlja vjerojatnost suda."""
        self.vjerojatnosti[sud] = vjerojatnost

    def uvjetna(self, A, B):
        """Računa P(A|B) = P(A ∧ B) / P(B)."""
        p_a_i_b = self.vjerojatnosti.get((A, B), 0)
        p_b = self.vjerojatnosti.get(B, 0)

        if p_b == 0:
            return None  # Nedefinirana
        return p_a_i_b / p_b

    def epistemoloski_opis(self, A, B, p_uvjetna):
        """Daje epistemološku interpretaciju uvjetne vjerojatnosti."""
        if p_uvjetna is None:
            return "Nedefinirana - nema znanja iz ovog uvjeta."
        elif p_uvjetna > self.vjerojatnosti.get(A, 0):
            return f"Dokaz '{B}' potvrđuje '{A}'."
        elif p_uvjetna < self.vjerojatnosti.get(A, 0):
            return f"Dokaz '{B}' oslabljuje '{A}'."
        else:
            return f"Dokaz '{B}' je neutralan za '{A}'."

# Primjer: Znanstvena teorija i eksperiment
print("Uvjetna vjerojatnost - Epistemološki primjer")
print("="*45)
print("\nKontekst: Znanstveno istraživanje\n")

uv = UvjetnaVjerojatnost("Znanstvena metoda")

# Postavi vjerojatnosti
uv.postavi("teorija", 0.3)  # Apriorna vjerojatnost teorije
uv.postavi("eksperiment", 0.4)  # Vjerojatnost pozitivnog eksperimenta
uv.postavi(("teorija", "eksperiment"), 0.25)  # Konjunkcija

print("Početne vjerojatnosti:")
print(f"  P(teorija istinita) = {uv.vjerojatnosti['teorija']:.2f}")
print(f"  P(pozitivan eksperiment) = {uv.vjerojatnosti['eksperiment']:.2f}")
print(f"  P(teorija istinita ∧ pozitivan eksperiment) = {uv.vjerojatnosti[('teorija',
↪  'eksperiment')]:.2f}")

print("\nUvjetne vjerojatnosti:\n")

# P(eksperiment | teorija)
p_eks_ako_teorija = uv.uvjetna("eksperiment", "teorija")
print("1. P(pozitivan eksperiment | teorija istinita)")
print(f"   = P(teorija ∧ eksperiment) / P(teorija)")
```

```
print(f"   = {uv.vjerojatnosti[('teorija', 'eksperiment')]:.2f} / {uv.vjerojatnosti['teorija']:.2f} = {p_eks_ako_teorija:.3f}")
print(f"   \n   Interpretacija: Ako je teorija istinita, eksperiment")
print(f"   će biti pozitivan u {p_eks_ako_teorija*100:.1f}% slučajeva.")

# P(teorija | eksperiment)
p_teorija_ako_eks = uv.uvjetna("teorija", "eksperiment")
print("\n2. P(teorija istinita | pozitivan eksperiment)")
print(f"   = P(teorija ∧ eksperiment) / P(eksperiment)")
print(f"   = {uv.vjerojatnosti[('teorija', 'eksperiment')]:.2f} / {uv.vjerojatnosti['eksperiment']:.2f} = {p_teorija_ako_eks:.3f}")
print(f"   \n   Interpretacija: Nakon pozitivnog eksperimenta,")
print(f"   vjerojatnost da je teorija istinita raste na {p_teorija_ako_eks*100:.1f}%.")

print("\nEpistemološka pouka:")
print("  Dokaz ne čini teoriju sigurnom, već povećava")
print("  stupanj racionalnog vjerovanja u nju.")
```

Output

```
Uvjetna vjerojatnost - Epistemološki primjer
============================================

Kontekst: Znanstveno istraživanje

Početne vjerojatnosti:
  P(teorija istinita) = 0.30
  P(pozitivan eksperiment) = 0.40
  P(teorija istinita ∧ pozitivan eksperiment) = 0.25

Uvjetne vjerojatnosti:

1. P(pozitivan eksperiment | teorija istinita)
   = P(teorija ∧ eksperiment) / P(teorija)
   = 0.25 / 0.30 = 0.000

   Interpretacija: Ako je teorija istinita, eksperiment
   će biti pozitivan u 0.0% slučajeva.

2. P(teorija istinita | pozitivan eksperiment)
   = P(teorija ∧ eksperiment) / P(eksperiment)
   = 0.25 / 0.40 = 0.625

   Interpretacija: Nakon pozitivnog eksperimenta,
   vjerojatnost da je teorija istinita raste na 62.5%.

Epistemološka pouka:
  Dokaz ne čini teoriju sigurnom, već povećava
  stupanj racionalnog vjerovanja u nju.
```

8.1.2 Bayesov teorem - formula racionalnog učenja

Bayesov teorem omogućava inverziju uvjetnih vjerojatnosti:

$$P(H|E) = \frac{P(E|H) \cdot P(H)}{P(E)}$$

gdje je:

- H - **hipoteza** (teorija, vjerovanje)
- E - **evidencija** (dokaz, opažanje)
- $P(H)$ - **prior** (apriorno vjerovanje)
- $P(H|E)$ - **posterior** (aposteriorno vjerovanje)
- $P(E|H)$ - **likelihood** (izglednost dokaza)

Ova formula opisuje **racionalno učenje** - kako ažurirati vjerovanja na temelju novih dokaza.

```python
import numpy as np
import matplotlib.pyplot as plt

class BayesovTeorem:
    """Implementacija i vizualizacija Bayesovog teorema."""

    def __init__(self):
        self.povijest_azuriranja = []

    def bayes(self, prior, likelihood, evidencija):
        """Primjenjuje Bayesov teorem.

        Args:
            prior: P(H)
            likelihood: P(E|H)
            evidencija: P(E)

        Returns:
            posterior: P(H|E)
        """
        if evidencija == 0:
            raise ValueError("P(E) ne može biti 0!")

        posterior = (likelihood * prior) / evidencija

        self.povijest_azuriranja.append({
            'prior': prior,
            'likelihood': likelihood,
```

```
            'evidencija': evidencija,
            'posterior': posterior
        })

        return posterior

    def bayes_omjer(self, prior1, prior2, likelihood1, likelihood2):
        """Računa Bayesov omjer za dvije hipoteze."""
        prior_omjer = prior1 / prior2 if prior2 > 0 else float('inf')
        likelihood_omjer = likelihood1 / likelihood2 if likelihood2 > 0 else float('inf')
        posterior_omjer = prior_omjer * likelihood_omjer

        return {
            'prior_omjer': prior_omjer,
            'likelihood_omjer': likelihood_omjer,
            'posterior_omjer': posterior_omjer
        }

# Demonstracija
print("Bayesov teorem - Izvod i značenje")
print("="*34)

print("\nMATEMATIČKI IZVOD:\n")
print("1. Definicija uvjetne vjerojatnosti:")
print("   P(H|E) = P(H ∧ E) / P(E)")
print("   P(E|H) = P(H ∧ E) / P(H)")
print("\n2. Iz druge jednadžbe:")
print("   P(H ∧ E) = P(E|H) · P(H)")
print("\n3. Uvrštavanjem u prvu:")
print("   P(H|E) = P(E|H) · P(H) / P(E)")

print("\nEPISTEMOLOŠKO ZNAČENJE:\n")
print("         P(E|H) · P(H)")
print("P(H|E) = -------------")
print("            P(E)")
print("\n↑            ↑       ↑        ↑")
print("Posterior  Likelihood Prior  Normalizacija")

print("\n• Prior P(H): Što sam vjerovao prije dokaza")
print("• Likelihood P(E|H): Koliko hipoteza predviđa dokaz")
print("• Posterior P(H|E): Što trebam vjerovati nakon dokaza")

print("\nALTERNATIVNI OBLIK (usporedba hipoteza):\n")
print("P(H₁|E)   P(E|H₁) · P(H₁)")
print("------- = ----------------")
print("P(H₂|E)   P(E|H₂) · P(H₂)")
print("\nOmjer posteriornih = Omjer likelihooda × Omjer priornih")

print("\nFILOZOFSKE IMPLIKACIJE:\n")
implikacije = [
    "Znanje je stupnjevito, ne binarno",
    "Učenje je ažuriranje, ne zamjena",
    "Dokazi ne govore sami - trebaju prior",
```

```
    "Racionalni agent mijenja mišljenje postupno"
]
for i, impl in enumerate(implikacije, 1):
    print(f"{i}. {impl}")
```

Output

```
Bayesov teorem - Izvod i značenje
==================================

MATEMATIČKI IZVOD:

1. Definicija uvjetne vjerojatnosti:
   P(H|E) = P(H ∧ E) / P(E)
   P(E|H) = P(H ∧ E) / P(H)

2. Iz druge jednadžbe:
   P(H ∧ E) = P(E|H) · P(H)

3. Uvrštavanjem u prvu:
   P(H|E) = P(E|H) · P(H) / P(E)

EPISTEMOLOŠKO ZNAČENJE:

          P(E|H) · P(H)
P(H|E) = -------------
             P(E)

↑            ↑       ↑        ↑
Posterior  Likelihood Prior  Normalizacija

• Prior P(H): Što sam vjerovao prije dokaza
• Likelihood P(E|H): Koliko hipoteza predviđa dokaz
• Posterior P(H|E): Što trebam vjerovati nakon dokaza

ALTERNATIVNI OBLIK (usporedba hipoteza):

P(H₁|E)   P(E|H₁) · P(H₁)
------- = ----------------
P(H₂|E)   P(E|H₂) · P(H₂)

Omjer posteriornih = Omjer likelihooda × Omjer priornih

FILOZOFSKE IMPLIKACIJE:

1. Znanje je stupnjevito, ne binarno
2. Učenje je ažuriranje, ne zamjena
3. Dokazi ne govore sami - trebaju prior
4. Racionalni agent mijenja mišljenje postupno
```

8.1.3 Epistemologija - znanje kao ažuriranje vjerovanja

Bayesov pristup revolucionira epistemologiju:

- **Tradicionalna epistemologija**: Znanje = opravdano istinito vjerovanje (JTB)
- **Bayesova epistemologija**: Znanje = racionalno ažurirano vjerovanje

"Mijenjanje mišljenja je dokaz razmišljanja" - preuređeni Bayesov princip

```python
class BayesovaEpistemologija:
    """Modelira epistemološki proces kroz Bayesovo ažuriranje."""

    def __init__(self, hipoteza, prior):
        self.hipoteza = hipoteza
        self.prior = prior
        self.trenutno_vjerovanje = prior
        self.povijest = [("početno", prior)]

    def promatraj_dokaz(self, dokaz, likelihood_true, likelihood_false):
        """Ažurira vjerovanje na temelju novog dokaza.

        Args:
            dokaz: opis dokaza
            likelihood_true: P(dokaz|hipoteza)
            likelihood_false: P(dokaz|¬hipoteza)
        """
        # Računaj P(dokaz)
        p_dokaz = (likelihood_true * self.trenutno_vjerovanje +
                   likelihood_false * (1 - self.trenutno_vjerovanje))

        # Bayesovo ažuriranje
        novo_vjerovanje = (likelihood_true * self.trenutno_vjerovanje) / p_dokaz

        promjena = novo_vjerovanje - self.trenutno_vjerovanje
        self.trenutno_vjerovanje = novo_vjerovanje
        self.povijest.append((dokaz, novo_vjerovanje))

        return {
            'novo_vjerovanje': novo_vjerovanje,
            'promjena': promjena,
            'p_dokaz': p_dokaz
        }

    def epistemoloski_status(self):
        """Vraća epistemološki status hipoteze."""
        v = self.trenutno_vjerovanje
        if v < 0.01:
            return "Praktički opovrgnuto"
```

```
        elif v < 0.1:
            return "Vrlo nevjerojatno"
        elif v < 0.4:
            return "Nevjerojatno"
        elif v < 0.6:
            return "Neizvjesno"
        elif v < 0.9:
            return "Vjerojatno"
        elif v < 0.99:
            return "Vrlo vjerojatno"
        else:
            return "Praktički sigurno"

# Simulacija znanstvenog istraživanja
print("Bayesova epistemologija - Ažuriranje znanja")
print("="*44)
print("\nScenarij: Znanstvenik istražuje novu teoriju\n")

# Početno stanje
episteme = BayesovaEpistemologija("Nova teorija", prior=0.1)
print("Početno stanje:")
print(f"  Prior P(teorija) = {episteme.prior:.3f}")
print(f"  Skepticizam: Visok ({(1-episteme.prior)*100:.1f}%)")

# Serija eksperimenata
eksperimenti = [
    ("POZITIVAN", 0.8, 0.3),   # Pozitivan rezultat
    ("POZITIVAN", 0.8, 0.3),   # Još jedan pozitivan
    ("NEGATIVAN", 0.2, 0.7),   # Negativan rezultat
    ("POZITIVAN", 0.8, 0.3),   # Opet pozitivan
    ("POZITIVAN", 0.8, 0.3),   # I još jedan
]

for i, (rezultat, lik_true, lik_false) in enumerate(eksperimenti, 1):
    print(f"\nEKSPERIMENT {i}:")
    print(f"  Rezultat: {rezultat}")

    if rezultat == "POZITIVAN":
        print(f"  Likelihood P(pozitivan|teorija) = {lik_true:.2f}")
        print(f"  P(pozitivan|¬teorija) = {lik_false:.2f}")
    else:
        print(f"  Likelihood P(negativan|teorija) = {lik_true:.2f}")
        print(f"  P(negativan|¬teorija) = {lik_false:.2f}")

    # Ažuriraj vjerovanje
    rezultat_azuriranja = episteme.promatraj_dokaz(
        f"Eksperiment {i}: {rezultat}",
        lik_true,
        lik_false
    )

    print(f"  \n  Bayesovo ažuriranje:")
    if i == 1:
```

```
        print(f"  P(teorija|dokaz₁) = {lik_true:.2f} × {episteme.povijest[-2][1]:.2f} / "
              f"{rezultat_azuriranja['p_dokaz']:.2f} =
              ↪ {rezultat_azuriranja['novo_vjerovanje']:.3f}")
    else:
        print(f"  P(teorija|{'dokaz₁' if i==2 else 'dokaz₁,dokaz₂' if i==3 else
        ↪  'dokaz₁,dokaz₂,dokaz₃' if i==4 else 'svi dokazi'}) = "
              f"{lik_true:.2f} × {episteme.povijest[-2][1]:.2f} / "
              f"{rezultat_azuriranja['p_dokaz']:.2f} =
              ↪ {rezultat_azuriranja['novo_vjerovanje']:.3f}")

    print(f"  \n  Novo vjerovanje: {rezultat_azuriranja['novo_vjerovanje']*100:.1f}%")
    print(f"  Promjena: {rezultat_azuriranja['promjena']*100:+.1f} postotnih bodova")

# Analiza
print("\nEpistemološka analiza:")
print("="*23)

print("\nPutanja vjerovanja: ", end="")
for i, (opis, vjerovanje) in enumerate(episteme.povijest):
    if i > 0:
        print(" → ", end="")
    print(f"{vjerovanje*100:.1f}%", end="")
print()

print("\nKljučne točke:")
print("  • Znanje se akumulira postupno")
print("  • Negativni dokazi također informiraju")
print("  • Nikad ne dostižemo potpunu sigurnost")
print("  • Racionalni agent mijenja mišljenje s dokazima")

print("\nFinalno stanje:")
print(f"  Vjerovanje u teoriju: {episteme.trenutno_vjerovanje*100:.1f}%")
print(f"  Status: {episteme.epistemoloski_status()} ", end="")
if episteme.trenutno_vjerovanje > 0.5:
    print("(više vjerojatno nego ne)")
else:
    print("(više nevjerojatno nego vjerojatno)")
```

Output

```
Bayesova epistemologija - Ažuriranje znanja
============================================

Scenarij: Znanstvenik istražuje novu teoriju

Početno stanje:
  Prior P(teorija) = 0.100
  Skepticizam: Visok (90.0%)

EKSPERIMENT 1:
  Rezultat: POZITIVAN
```

```
  Likelihood P(pozitivan|teorija) = 0.80
  P(pozitivan|¬teorija) = 0.30

  Bayesovo ažuriranje:
  P(teorija|dokaz_1) = 0.80 × 0.10 / 0.35 = 0.229

  Novo vjerovanje: 22.9%
  Promjena: +12.9 postotnih bodova

EKSPERIMENT 2:
  Rezultat: POZITIVAN
  Likelihood P(pozitivan|teorija) = 0.80
  P(pozitivan|¬teorija) = 0.30

  Bayesovo ažuriranje:
  P(teorija|dokaz_1) = 0.80 × 0.23 / 0.41 = 0.441

  Novo vjerovanje: 44.1%
  Promjena: +21.3 postotnih bodova

EKSPERIMENT 3:
  Rezultat: NEGATIVAN
  Likelihood P(negativan|teorija) = 0.20
  P(negativan|¬teorija) = 0.70

  Bayesovo ažuriranje:
  P(teorija|dokaz_1,dokaz_2) = 0.20 × 0.44 / 0.48 = 0.184

  Novo vjerovanje: 18.4%
  Promjena: -25.7 postotnih bodova

EKSPERIMENT 4:
  Rezultat: POZITIVAN
  Likelihood P(pozitivan|teorija) = 0.80
  P(pozitivan|¬teorija) = 0.30

  Bayesovo ažuriranje:
  P(teorija|dokaz_1,dokaz_2,dokaz_3) = 0.80 × 0.18 / 0.39 = 0.376

  Novo vjerovanje: 37.6%
  Promjena: +19.2 postotnih bodova

EKSPERIMENT 5:
  Rezultat: POZITIVAN
  Likelihood P(pozitivan|teorija) = 0.80
  P(pozitivan|¬teorija) = 0.30

  Bayesovo ažuriranje:
  P(teorija|svi dokazi) = 0.80 × 0.38 / 0.49 = 0.616
```

```
  Novo vjerovanje: 61.6%
  Promjena: +24.0 postotnih bodova

Epistemološka analiza:
======================

Putanja vjerovanja: 10.0% → 22.9% → 44.1% → 18.4% → 37.6% → 61.6%

Ključne točke:
  • Znanje se akumulira postupno
  • Negativni dokazi također informiraju
  • Nikad ne dostižemo potpunu sigurnost
  • Racionalni agent mijenja mišljenje s dokazima

Finalno stanje:
  Vjerovanje u teoriju: 61.6%
  Status: Vjerojatno (više vjerojatno nego ne)
```

8.1.4 Filozofija znanosti - potvrđivanje i opovrgavanje

Bayesov pristup osvjetljava klasične debate u filozofiji znanosti:

Popper vs. Bayes

- **Popper**: Znanosti se bavi **falsifikacijom** - traži opovrgavanje
- **Bayes**: Znanost postupno **ažurira vjerojatnosti** - traži najbolju hipotezu

Problem indukcije

> "Nema dovoljno opažanja bijele boje koje bi dokazalo da su svi labudovi bijeli, ali jedno opažanje crnog labuda to opovrgava" - Karl Popper

Bayesov pristup: Crni labud drastično smanjuje vjerojatnost, ali ne na nulu!

```python
class FilozofijaZnanosti:
    """Uspoređuje Popperov i Bayesov pristup znanstvenoj metodi."""

    def __init__(self):
        self.popper_status = "nije opovrgnuta"
        self.bayes_vjerovanje = {}

    def popper_test(self, hipoteza, opazanje, konzistentno):
        """Popperov pristup - falsifikacija."""
```

```
        if not konzistentno:
            self.popper_status = "OPOVRGNUTA"
            return False
        return True  # Nije opovrgnuta (ali nije ni potvrđena!)

    def bayes_test(self, hipoteze, opazanje, likelihoods):
        """Bayesov pristup - ažuriranje vjerojatnosti."""
        # Računaj P(opažanje)
        p_opazanje = sum(likelihoods[h] * self.bayes_vjerovanje[h]
                         for h in hipoteze)

        if p_opazanje == 0:
            return self.bayes_vjerovanje

        # Ažuriraj sve hipoteze
        nova_vjerovanja = {}
        for h in hipoteze:
            nova_vjerovanja[h] = (likelihoods[h] * self.bayes_vjerovanje[h]) / p_opazanje

        self.bayes_vjerovanje = nova_vjerovanja
        return nova_vjerovanja

# Simulacija: Problem bijelih labudova
print("Filozofija znanosti - Popper vs. Bayes")
print("="*39)
print("\nHipoteza: 'Svi labudovi su bijeli'\n")

fz = FilozofijaZnanosti()

# POPPEROV PRISTUP
print("POPPEROV PRISTUP (Falsifikacija):")
print("-"*35)

# 1000 bijelih labudova
for i in [1, 2, 3, "...", 1000]:
    if i == "...":
        print(f"... (nakon 999 bijelih labudova)")
    else:
        status = fz.popper_test("svi bijeli", "bijeli", True)
        print(f"Opažanje {i}: Bijeli labud → Hipoteza NIJE OPOVRGNUTA")

print(f"\nStatus: Hipoteza {fz.popper_status} (ali nije ni potvrđena!)")

# Crni labud!
print(f"\nOpažanje 1001: CRNI LABUD → Hipoteza OPOVRGNUTA! ×")
fz.popper_test("svi bijeli", "crni", False)

print("\nPopperov zaključak: Hipoteza je falsificirana.")
print("Tisuću potvrda ne dokazuje teoriju, jedna falsifikacija je ruši.")

# BAYESOV PRISTUP
print("\nBAYESOV PRISTUP (Ažuriranje vjerojatnosti):")
print("-"*44)
```

```
63
64 # Početna vjerovanja
65 fz.bayes_vjerovanje = {
66     "svi_bijeli": 0.5,
67     "vecina_bijeli": 0.5
68 }
69 print(f"Početno vjerovanje: P(svi bijeli) = {fz.bayes_vjerovanje['svi_bijeli']*100:.1f}%\n")
70
71 # Likelihoods za bijeli labud
72 lik_bijeli = {
73     "svi_bijeli": 1.0,      # Ako su svi bijeli, sigurno ćemo vidjeti bijelog
74     "vecina_bijeli": 0.9    # Ako je većina bijela, 90% šanse za bijelog
75 }
76
77 # Simuliraj opažanja
78 brojevi_opazanja = [10, 100, 500, 1000]
79 for n in brojevi_opazanja:
80     # Resetiraj za svaki test
81     fz.bayes_vjerovanje = {"svi_bijeli": 0.5, "vecina_bijeli": 0.5}
82
83     for _ in range(n):
84         fz.bayes_test(["svi_bijeli", "vecina_bijeli"], "bijeli", lik_bijeli)
85
86     print(f"Nakon {n} bijelih: P(hipoteza) = {fz.bayes_vjerovanje['svi_bijeli']*100:.1f}%")
87
88 # Crni labud!
89 print("\nOpažanje 1001: CRNI LABUD!")
90 lik_crni = {
91     "svi_bijeli": 0.0,      # Nemoguće ako su svi bijeli!
92     "vecina_bijeli": 0.1    # Moguće ako neki nisu bijeli
93 }
94 print(f"Likelihood P(crni|svi bijeli) = {lik_crni['svi_bijeli']:.3f}")
95 print(f"Likelihood P(crni|neki crni) = {lik_crni['vecina_bijeli']:.3f}")
96
97 fz.bayes_test(["svi_bijeli", "vecina_bijeli"], "crni", lik_crni)
98 print(f"\nNakon crnog labuda: P(hipoteza) = {fz.bayes_vjerovanje['svi_bijeli']*100:.1f}%")
99 print(f"Alternativa P(većina bijeli) = {fz.bayes_vjerovanje['vecina_bijeli']*100:.1f}%")
100
101 print("\nBayesov zaključak: Prebacujemo vjerovanje na fleksibilniju hipotezu.")
102
103 # Usporedba
104 print("\nUSPOREDBA PRISTUPA:")
105 print("="*20)
106
107 print("\nPopper:")
108 print("  ✓ Jasan kriterij (opovrgnuto/nije)")
109 print("  ✓ Naglašava kritičko testiranje")
110 print("  × Binaran pristup (sve ili ništa)")
111 print("  × Ne govori o stupnju potvrde")
112
113 print("\nBayes:")
114 print("  ✓ Stupnjevito znanje")
115 print("  ✓ Kvantificira nesigurnost")
```

```
116 print("  ✓ Omogućava usporedbu hipoteza")
117 print("  × Ovisi o prioru")
118 print("  × Složeniji račun")
119
120 print("\nSinteza: Popperova falsifikacija je specijalan slučaj")
121 print("         Bayesovog ažuriranja s likelihood = 0")
```

Output

```
Filozofija znanosti - Popper vs. Bayes
======================================

Hipoteza: 'Svi labudovi su bijeli'

POPPEROV PRISTUP (Falsifikacija):
---------------------------------
Opažanje 1: Bijeli labud → Hipoteza NIJE OPOVRGNUTA
Opažanje 2: Bijeli labud → Hipoteza NIJE OPOVRGNUTA
Opažanje 3: Bijeli labud → Hipoteza NIJE OPOVRGNUTA
... (nakon 999 bijelih labudova)
Opažanje 1000: Bijeli labud → Hipoteza NIJE OPOVRGNUTA

Status: Hipoteza nije opovrgnuta (ali nije ni potvrđena!)

Opažanje 1001: CRNI LABUD → Hipoteza OPOVRGNUTA! ×

Popperov zaključak: Hipoteza je falsificirana.
Tisuću potvrda ne dokazuje teoriju, jedna falsifikacija je ruši.

BAYESOV PRISTUP (Ažuriranje vjerojatnosti):
-------------------------------------------
Početno vjerovanje: P(svi bijeli) = 50.0%

Nakon 10 bijelih: P(hipoteza) = 74.1%
Nakon 100 bijelih: P(hipoteza) = 100.0%
Nakon 500 bijelih: P(hipoteza) = 100.0%
Nakon 1000 bijelih: P(hipoteza) = 100.0%

Opažanje 1001: CRNI LABUD!
Likelihood P(crni|svi bijeli) = 0.000
Likelihood P(crni|neki crni) = 0.100

Nakon crnog labuda: P(hipoteza) = 0.0%
Alternativa P(većina bijeli) = 100.0%

Bayesov zaključak: Prebacujemo vjerovanje na fleksibilniju hipotezu.

USPOREDBA PRISTUPA:
===================
```

```
Popper:
  ✓ Jasan kriterij (opovrgnuto/nije)
  ✓ Naglašava kritičko testiranje
  × Binaran pristup (sve ili ništa)
  × Ne govori o stupnju potvrde

Bayes:
  ✓ Stupnjevito znanje
  ✓ Kvantificira nesigurnost
  ✓ Omogućava usporedbu hipoteza
  × Ovisi o prioru
  × Složeniji račun

Sinteza: Popperova falsifikacija je specijalan slučaj
         Bayesovog ažuriranja s likelihood = 0
```

8.1.5 Problem priornih vjerojatnosti

Najkontroverzniji aspekt Bayesove epistemologije je izbor **priora**:

- **Subjektivni prior**: Osobno vjerovanje
- **Objektivni prior**: Princip indiferencije, maksimalna entropija
- **Informativni prior**: Na temelju prethodnog znanja

> "Prior je mjesto gdje se subjektivnost uvlači u objektivnu znanost" - kritičari Bayesa

```python
class ProblemPriora:
    """Istražuje epistemološke implikacije izbora priora."""

    def __init__(self):
        self.agenti = {}

    def stvori_agenta(self, ime, prior):
        """Stvara epistemološkog agenta s danim priorom."""
        self.agenti[ime] = {
            'prior': prior,
            'trenutno': prior,
            'povijest': [prior]
        }

    def azuriraj_sve(self, likelihood_h, likelihood_ne_h):
        """Svi agenti promatraju isti dokaz."""
        rezultati = {}

```

```
        for ime, agent in self.agenti.items():
            p_trenutno = agent['trenutno']

            # Bayesovo ažuriranje
            p_dokaz = likelihood_h * p_trenutno + likelihood_ne_h * (1 - p_trenutno)
            p_novo = (likelihood_h * p_trenutno) / p_dokaz if p_dokaz > 0 else p_trenutno

            agent['trenutno'] = p_novo
            agent['povijest'].append(p_novo)

            rezultati[ime] = {
                'prije': p_trenutno,
                'poslije': p_novo,
                'promjena': p_novo - p_trenutno
            }

        return rezultati

    def mjeri_konvergenciju(self):
        """Mjeri koliko su se vjerovanja približila."""
        trenutna_vjerovanja = [a['trenutno'] for a in self.agenti.values()]
        return max(trenutna_vjerovanja) - min(trenutna_vjerovanja)

# Demonstracija
print("Problem priora - Epistemološka kontroverza")
print("="*43)
print("\nScenarij: Tri znanstvenika procjenjuju istu hipotezu")
print("         nakon identičnih dokaza\n")

pp = ProblemPriora()

# Različiti priori
pp.stvori_agenta("Optimist", 0.8)
pp.stvori_agenta("Neutralac", 0.5)
pp.stvori_agenta("Skeptik", 0.1)

print("POČETNI PRIORI:")
for ime, agent in pp.agenti.items():
    prior_opis = "(visoko vjerovanje)" if agent['prior'] > 0.6 else \
                 "(princip indiferencije)" if agent['prior'] == 0.5 else \
                 "(nisko vjerovanje)"
    print(f"  {ime:12} P(H) = {agent['prior']:.2f} {prior_opis}")

# Serija dokaza
dokazi = [
    ("Pozitivan", 0.9, 0.2),
    ("Pozitivan", 0.9, 0.2),
    ("Negativan", 0.1, 0.8),
    ("Pozitivan", 0.9, 0.2),
    ("Pozitivan", 0.9, 0.2),
]

for i, (tip, lik_h, lik_ne_h) in enumerate(dokazi, 1):
```

```
    print(f"\nDOKAZ {i}: {tip} (P(E|H)={lik_h}, P(E|¬H)={lik_ne_h})")

    rezultati = pp.azuriraj_sve(lik_h, lik_ne_h)

    print("Nakon dokaza:")
    for ime, rez in rezultati.items():
        print(f"  {ime:12} {rez['prije']:.2f} → {rez['poslije']:.2f} "
              f"({rez['promjena']:+.2f})")

# Analiza konvergencije
print("\nANALIZA KONVERGENCIJE:")
print("="*23)

pocetna_razlika = max(a['prior'] for a in pp.agenti.values()) - \
                  min(a['prior'] for a in pp.agenti.values())
finalna_razlika = pp.mjeri_konvergenciju()

print(f"\nPočetna razlika (max-min): {pocetna_razlika:.2f}")
print(f"Finalna razlika (max-min): {finalna_razlika:.2f}")

print("\nFinalna vjerovanja:")
for ime, agent in pp.agenti.items():
    print(f"  {ime:10} {agent['trenutno']*100:.1f}%")

print("\nEpistemološke implikacije:")
print("  • Priori utječu, ali njihov utjecaj slabi s dokazima")
print("  • Dovoljno dokaza vodi konvergenciji")
print("  • Različiti putovi do (približno) istog zaključka")
print("  • Objektivnost emerge iz subjektivnih početaka")

print("\nFILOZOFSKI PROBLEM:")
print("  Koji prior je 'pravi'? Postoji li objektivan prior?")
print("  \n  Pristupi:")
print("  1. Princip indiferencije (Laplace): P=0.5 bez informacija")
print("  2. Jeffreysov prior: Invarijantan na reparametrizaciju  ")
print("  3. Maksimalna entropija: Najmanje informativna distribucija")
print("  4. Empirijski Bayes: Prior iz podataka")
print("  \n  Zaključak: Prior je neizbježan, ali ne fatalan!")
```

Output

```
Problem priora - Epistemološka kontroverza
==========================================

Scenarij: Tri znanstvenika procjenjuju istu hipotezu
          nakon identičnih dokaza

POČETNI PRIORI:
  Optimist     P(H) = 0.80 (visoko vjerovanje)
  Neutralac    P(H) = 0.50 (princip indiferencije)
  Skeptik      P(H) = 0.10 (nisko vjerovanje)
```

```
DOKAZ 1: Pozitivan (P(E|H)=0.9, P(E|¬H)=0.2)
Nakon dokaza:
  Optimist     0.80 → 0.95 (+0.15)
  Neutralac    0.50 → 0.82 (+0.32)
  Skeptik      0.10 → 0.33 (+0.23)

DOKAZ 2: Pozitivan (P(E|H)=0.9, P(E|¬H)=0.2)
Nakon dokaza:
  Optimist     0.95 → 0.99 (+0.04)
  Neutralac    0.82 → 0.95 (+0.13)
  Skeptik      0.33 → 0.69 (+0.36)

DOKAZ 3: Negativan (P(E|H)=0.1, P(E|¬H)=0.8)
Nakon dokaza:
  Optimist     0.99 → 0.91 (-0.08)
  Neutralac    0.95 → 0.72 (-0.24)
  Skeptik      0.69 → 0.22 (-0.47)

DOKAZ 4: Pozitivan (P(E|H)=0.9, P(E|¬H)=0.2)
Nakon dokaza:
  Optimist     0.91 → 0.98 (+0.07)
  Neutralac    0.72 → 0.92 (+0.20)
  Skeptik      0.22 → 0.56 (+0.34)

DOKAZ 5: Pozitivan (P(E|H)=0.9, P(E|¬H)=0.2)
Nakon dokaza:
  Optimist     0.98 → 1.00 (+0.02)
  Neutralac    0.92 → 0.98 (+0.06)
  Skeptik      0.56 → 0.85 (+0.29)

ANALIZA KONVERGENCIJE:
======================

Početna razlika (max-min): 0.70
Finalna razlika (max-min): 0.14

Finalna vjerovanja:
  Optimist   99.5%
  Neutralac  98.1%
  Skeptik    85.1%

Epistemološke implikacije:
  • Priori utječu, ali njihov utjecaj slabi s dokazima
  • Dovoljno dokaza vodi konvergenciji
  • Različiti putovi do (približno) istog zaključka
  • Objektivnost emerge iz subjektivnih početaka

FILOZOFSKI PROBLEM:
  Koji prior je 'pravi'? Postoji li objektivan prior?
```

```
Pristupi:
1. Princip indiferencije (Laplace): P=0.5 bez informacija
2. Jeffreysov prior: Invarijantan na reparametrizaciju
3. Maksimalna entropija: Najmanje informativna distribucija
4. Empirijski Bayes: Prior iz podataka

Zaključak: Prior je neizbježan, ali ne fatalan!
```

8.1.6 Occamova oštrica i Bayesov faktor

Occamova oštrica: "Entitete ne treba umnožavati bez potrebe"

Bayesov pristup prirodno implementira ovaj princip kroz **Bayesov faktor**:

$$BF_{12} = \frac{P(E|H_1)}{P(E|H_2)} \cdot \frac{P(H_1)}{P(H_2)}$$

Jednostavnije hipoteze imaju veću priornu vjerojatnost i manju fleksibilnost u objašnjavanju podataka.

```python
class OccamovaOstrica:
    """Implementira Occamovu oštricu kroz Bayesov faktor."""

    def __init__(self):
        self.hipoteze = {}

    def dodaj_hipotezu(self, naziv, kompleksnost, prior=None):
        """Dodaje hipotezu s danom kompleksnošću.

        Prior se automatski računa prema kompleksnosti ako nije dan.
        """
        if prior is None:
            # Occamov princip: jednostavnije hipoteze imaju veći prior
            prior = 1.0 / (2 ** kompleksnost)

        self.hipoteze[naziv] = {
            'kompleksnost': kompleksnost,
            'prior': prior,
            'posterior': prior
        }

    def bayesov_faktor(self, h1, h2, likelihood1, likelihood2):
        """Računa Bayesov faktor između dvije hipoteze."""
        prior_omjer = self.hipoteze[h1]['prior'] / self.hipoteze[h2]['prior']
        likelihood_omjer = likelihood1 / likelihood2 if likelihood2 > 0 else float('inf')

        bf = likelihood_omjer * prior_omjer
```

```
28
29          return {
30              'prior_omjer': prior_omjer,
31              'likelihood_omjer': likelihood_omjer,
32              'bayesov_faktor': bf
33          }
34
35      def interpretacija_bf(self, bf):
36          """Interpretira Bayesov faktor prema Jeffreys skali."""
37          if bf < 1:
38              return "Dokaz protiv H₁"
39          elif bf < 3:
40              return "Slab dokaz za H₁"
41          elif bf < 10:
42              return "Umjeren dokaz za H₁"
43          elif bf < 100:
44              return "Jak dokaz za H₁"
45          else:
46              return "Odlučujući dokaz za H₁"
47
48  # Demonstracija
49  print("Occamova oštrica kroz Bayesov faktor")
50  print("="*37)
51  print("\nScenarij: Objašnjenje niza brojeva [2, 4, 6, 8, 10]\n")
52
53  oo = OccamovaOstrica()
54
55  # Dvije hipoteze različite složenosti
56  print("HIPOTEZE:")
57  print("  H₁ (jednostavna): 'Parni brojevi do 10'")
58  print("      Parametri: 1 (samo parnost)")
59  oo.dodaj_hipotezu("H1_jednostavna", kompleksnost=1, prior=0.67)
60
61  print("      ")
62  print("  H₂ (složena): 'Brojevi koji zadovoljavaju x²-12x+20<0 ili x=2k, k∈ℕ'")
63  print("      Parametri: 5 (polinomski koeficijenti + parnost)")
64  oo.dodaj_hipotezu("H2_slozena", kompleksnost=5, prior=0.33)
65
66  print("\nPRIORNE VJEROJATNOSTI (Occamov princip):")
67  print(f"  P(H₁) = {oo.hipoteze['H1_jednostavna']['prior']:.2f} (preferiramo jednostavnost)")
68  print(f"  P(H₂) = {oo.hipoteze['H2_slozena']['prior']:.2f} (penaliziramo složenost)")
69
70  prior_omjer = oo.hipoteze['H1_jednostavna']['prior'] / oo.hipoteze['H2_slozena']['prior']
71  print(f"  \n  Prior omjer: P(H₁)/P(H₂) = {prior_omjer:.2f}")
72
73  # Test predviđanja
74  print("\nTEST 1: Predviđanje sljedećeg broja")
75  print("  H₁ predviđa: 12 (sljedeći parni)")
76  print("  H₂ predviđa: 12 ili 7.3 (fleksibilnija)")
77  print("  \n  Opažanje: 12")
78
79  # Likelihoods
80  lik_h1 = 1.0  # H1 savršeno predviđa 12
```

```
lik_h2 = 0.5  # H2 daje 50% šanse za 12

print(f"  P(12|H₁) = {lik_h1:.2f} (savršeno predviđanje)")
print(f"  P(12|H₂) = {lik_h2:.2f} (jedan od mogućih)")

bf_rezultat = oo.bayesov_faktor("H1_jednostavna", "H2_slozena", lik_h1, lik_h2)
print(f"  \n  Likelihood omjer: {bf_rezultat['likelihood_omjer']:.2f}")
print(f"  Bayesov faktor: BF₁₂ = {bf_rezultat['likelihood_omjer']:.2f} × {prior_omjer:.2f} =
↪  {bf_rezultat['bayesov_faktor']:.2f}")
print(f"  \n  Interpretacija: Dokazi {bf_rezultat['bayesov_faktor']:.0f}:1 u korist H₁")

# Drugi test
print("\nTEST 2: Još jedno predviđanje")
print("  Opažanje: 14")
print(f"  P(14|H₁) = 1.00")
print(f"  P(14|H₂) = 0.50")

# Kumulativni BF
kumulativni_bf = bf_rezultat['bayesov_faktor'] * 2  # još jedan faktor 2
print(f"  \n  Kumulativni BF₁₂ = {kumulativni_bf:.2f}")
print(f"  \n  Interpretacija: Dokazi {kumulativni_bf:.0f}:1 u korist H₁")

# Filozofska analiza
print("\nFILOZOFSKA ANALIZA:")
print("="*20)
print("\nZašto Bayes preferira jednostavnost?\n")

print("1. PRIOR KOMPONENTA:")
print("   Jednostavnije hipoteze su a priori vjerovatnije")
print("   (manje parametara = veća gustoća vjerojatnosti)")

print("\n2. LIKELIHOOD KOMPONENTA:")
print("   Složene hipoteze 'razmazuju' vjerojatnost")
print("   na više mogućnosti → niži likelihood po ishodu")

print("\n3. AUTOMATSKA PENALIZACIJA:")
print("   Prefleksibilne teorije se same kažnjavaju")
print("   jer mogu objasniti previše → slabo predviđaju")

print("\nJEFFREYS-OVA SKALA za interpretaciju BF:")
skala = [
    ("BF < 1", "Dokaz protiv H₁"),
    ("1 < BF < 3", "Slab dokaz za H₁"),
    ("3 < BF < 10", "Umjeren dokaz za H₁"),
    ("BF > 10", "Jak dokaz za H₁"),
    ("BF > 100", "Odlučujući dokaz za H₁")
]
for raspon, opis in skala:
    print(f"  {raspon:11}: {opis}")

print(f"\nNaš slučaj: BF = {kumulativni_bf:.2f} → {oo.interpretacija_bf(kumulativni_bf)}")

print("\n> \"Priroda je jednostavna i ne umnožava uzroke bez potrebe\" - Newton")
```

```
133 print("> Bayesov faktor to kvantificira!")
```

Output

```
Occamova oštrica kroz Bayesov faktor
====================================

Scenarij: Objašnjenje niza brojeva [2, 4, 6, 8, 10]

HIPOTEZE:
  H₁ (jednostavna): 'Parni brojevi do 10'
      Parametri: 1 (samo parnost)

  H₂ (složena): 'Brojevi koji zadovoljavaju x²-12x+20<0 ili x=2k, k∈ℕ'
      Parametri: 5 (polinomski koeficijenti + parnost)

PRIORNE VJEROJATNOSTI (Occamov princip):
  P(H₁) = 0.67 (preferiramo jednostavnost)
  P(H₂) = 0.33 (penaliziramo složenost)

  Prior omjer: P(H₁)/P(H₂) = 2.03

TEST 1: Predviđanje sljedećeg broja
  H₁ predviđa: 12 (sljedeći parni)
  H₂ predviđa: 12 ili 7.3 (fleksibilnija)

  Opažanje: 12
  P(12|H₁) = 1.00 (savršeno predviđanje)
  P(12|H₂) = 0.50 (jedan od mogućih)

  Likelihood omjer: 2.00
  Bayesov faktor: BF₁₂ = 2.00 × 2.03 = 4.06

  Interpretacija: Dokazi 4:1 u korist H₁

TEST 2: Još jedno predviđanje
  Opažanje: 14
  P(14|H₁) = 1.00
  P(14|H₂) = 0.50

  Kumulativni BF₁₂ = 8.12

  Interpretacija: Dokazi 8:1 u korist H₁

FILOZOFSKA ANALIZA:
===================

Zašto Bayes preferira jednostavnost?

1. PRIOR KOMPONENTA:
```

```
   Jednostavnije hipoteze su a priori vjerovatnije
   (manje parametara = veća gustoća vjerojatnosti)

2. LIKELIHOOD KOMPONENTA:
   Složene hipoteze 'razmazuju' vjerojatnost
   na više mogućnosti → niži likelihood po ishodu

3. AUTOMATSKA PENALIZACIJA:
   Prefleksibilne teorije se same kažnjavaju
   jer mogu objasniti previše → slabo predviđaju

JEFFREYS-OVA SKALA za interpretaciju BF:
  BF < 1     : Dokaz protiv H₁
  1 < BF < 3 : Slab dokaz za H₁
  3 < BF < 10: Umjeren dokaz za H₁
  BF > 10    : Jak dokaz za H₁
  BF > 100   : Odlučujući dokaz za H₁

Naš slučaj: BF = 8.12 → Umjeren dokaz za H₁

> "Priroda je jednostavna i ne umnožava uzroke bez potrebe" - Newton
> Bayesov faktor to kvantificira!
```

8.1.7 Koherentnost i Dutch Book argument

Dutch Book teorem: Ako vaša vjerovanja krše aksiome vjerojatnosti, netko može konstruirati niz oklada gdje sigurno gubite novac.

Ovo daje **pragmatičko opravdanje** Bayesovog pristupa - nekoherentna vjerovanja vode u sigurni gubitak!

```python
class DutchBook:
    """Demonstrira Dutch Book argument protiv nekoherentnih vjerovanja."""

    def __init__(self, vjerovanja):
        """
        vjerovanja: dict s vjerojatnostima za različite sudove
        """
        self.vjerovanja = vjerovanja

    def provjeri_koherentnost(self):
        """Provjerava krše li vjerovanja aksiome vjerojatnosti."""
        problemi = []

        # Provjeri nenegativnost
        for sud, p in self.vjerovanja.items():
            if p < 0 or p > 1:
                problemi.append(f"{sud}: P={p} nije u [0,1]")
```

```
18
19          # Provjeri aditivnost za disjunkciju (ako postoji)
20          if 'A' in self.vjerovanja and 'B' in self.vjerovanja and 'A_ili_B' in
            ↪  self.vjerovanja:
21              p_a = self.vjerovanja['A']
22              p_b = self.vjerovanja['B']
23              p_a_ili_b = self.vjerovanja['A_ili_B']
24
25              # P(A ∨ B) ≤ P(A) + P(B)
26              if p_a_ili_b > p_a + p_b:
27                  problemi.append(f"P(A∨B)={p_a_ili_b} > P(A)+P(B)={p_a+p_b}")
28
29              # P(A ∨ B) ≥ max(P(A), P(B))
30              if p_a_ili_b < max(p_a, p_b):
31                  problemi.append(f"P(A∨B)={p_a_ili_b} < max(P(A),P(B))={max(p_a, p_b)}")
32
33          return len(problemi) == 0, problemi
34
35      def konstruiraj_dutch_book(self):
36          """Konstruira skup oklada koji garantira gubitak."""
37          if 'A' not in self.vjerovanja or 'B' not in self.vjerovanja or 'A_ili_B' not in
            ↪  self.vjerovanja:
38              return None
39
40          p_a = self.vjerovanja['A']
41          p_b = self.vjerovanja['B']
42          p_a_ili_b = self.vjerovanja['A_ili_B']
43
44          oklade = []
45
46          # Oklada 1: Kladi se na A po "fer" cijeni
47          oklade.append({
48              'tip': 'ZA',
49              'sud': 'A',
50              'cijena': p_a * 100,
51              'isplata': 100
52          })
53
54          # Oklada 2: Kladi se na B po "fer" cijeni
55          oklade.append({
56              'tip': 'ZA',
57              'sud': 'B',
58              'cijena': p_b * 100,
59              'isplata': 100
60          })
61
62          # Oklada 3: Kladi se PROTIV (A ili B)
63          oklade.append({
64              'tip': 'PROTIV',
65              'sud': 'A_ili_B',
66              'prima': p_a_ili_b * 100,
67              'placa_ako_istina': 100
68          })
```

```
69
70          return oklade
71
72  # Demonstracija
73  print("Dutch Book Argument - Cijena nekoherentnosti")
74  print("="*46)
75
76  # Nekoherentan agent
77  nekoherentan = DutchBook({
78      'A': 0.6,
79      'B': 0.5,
80      'A_ili_B': 0.7  # Prekršaj! Trebalo bi biti barem 0.6
81  })
82
83  print("\nAGENT 1: Nekoherentan")
84  print("Vjerovanja:")
85  for sud, p in nekoherentan.vjerovanja.items():
86      simbol = "∨" if "ili" in sud else ""
87      prikaz = sud.replace("_ili_", " ∨ ")
88      print(f"  P({prikaz}) = {p:.2f}", end="")
89      if sud == 'A_ili_B':
90          print(" KRŠI AKSIOME!")
91      else:
92          print()
93
94  koherentan_min = max(nekoherentan.vjerovanja['A'], nekoherentan.vjerovanja['B'])
95  koherentan_max = min(1.0, nekoherentan.vjerovanja['A'] + nekoherentan.vjerovanja['B'])
96  print(f"  \n  Trebalo bi biti: P(A ∨ B) ≥ max({nekoherentan.vjerovanja['A']:.2f},
    ↪ {nekoherentan.vjerovanja['B']:.2f}) = {koherentan_min:.2f}")
97  print(f"  i P(A ∨ B) ≤ {nekoherentan.vjerovanja['A']:.2f} +
    ↪ {nekoherentan.vjerovanja['B']:.2f} = {nekoherentan.vjerovanja['A'] +
    ↪ nekoherentan.vjerovanja['B']:.2f} (ali ≤ 1)")
98  print(f"  Koherentan raspon: [{koherentan_min:.2f}, {koherentan_max:.2f}]")
99
100 # Koherentan agent
101 koherentan = DutchBook({
102     'A': 0.6,
103     'B': 0.5,
104     'A_ili_B': 0.8  # Koherentno!
105 })
106
107 print("\nAGENT 2: Koherentan")
108 print("Vjerovanja:")
109 for sud, p in koherentan.vjerovanja.items():
110     prikaz = sud.replace("_ili_", " ∨ ")
111     print(f"  P({prikaz}) = {p:.2f}", end="")
112     if sud == 'A_ili_B':
113         print("  ✓ Zadovoljava aksiome")
114     else:
115         print()
116
117 # Konstruiraj Dutch Book
118 print("\nDUTCH BOOK PROTIV NEKOHERENTNOG AGENTA:")
```

```
print("="*41)

oklade = nekoherentan.konstruiraj_dutch_book()
print("\nKladionič ar konstruira sljedeće oklade:\n")

for i, oklada in enumerate(oklade, 1):
    if oklada['tip'] == 'ZA':
        print(f"Oklada {i}: Agent se kladi ZA '{oklada['sud']}'")
        print(f"  Fer cijena (prema agentu): {oklada['cijena']:.1f}€")
        print(f"  Dobitak ako {oklada['sud']}: {oklada['isplata']}€, Gubitak ako
        ↪  ¬{oklada['sud']}: -{oklada['cijena']:.0f}€")
    else:
        prikaz = oklada['sud'].replace("_ili_", " ∨ ")
        print(f"Oklada {i}: Agent se kladi PROTIV '{prikaz}'")
        print(f"  Agent prima: {oklada['prima']:.1f}€")
        print(f"  Mora platiti {oklada['placa_ako_istina']}€ ako {prikaz}")
    print()

# Analiza ishoda
print("ANALIZA ISHODA:")
print("="*16)
print("\nMogući ishodi:")

ishodi = [
    ("A∧B", True, True),
    ("A∧¬B", True, False),
    ("¬A∧B", False, True),
    ("¬A∧¬B", False, False)
]

for naziv, a_istina, b_istina in ishodi:
    profit = 0
    detalji = []

    # Oklada 1 (na A)
    if a_istina:
        profit += 100 - oklade[0]['cijena']
        detalji.append("+100")
    else:
        profit -= oklade[0]['cijena']
        detalji.append(f"-{oklade[0]['cijena']:.0f}")

    # Oklada 2 (na B)
    if b_istina:
        profit += 100 - oklade[1]['cijena']
        detalji.append("+100")
    else:
        profit -= oklade[1]['cijena']
        detalji.append(f"-{oklade[1]['cijena']:.0f}")

    # Oklada 3 (protiv A ili B)
    if a_istina or b_istina:
        profit += oklade[2]['prima'] - 100
```

```
        detalji.append("-100")
    else:
        profit += oklade[2]['prima']
        detalji.append("+0")

    print(f"  {naziv:8} {' '.join(detalji):15} = {profit:+.0f}€  → Agent {'gubi' if profit < 0 else 'dobiva'} {abs(profit):.0f}€")

print("\nAgent UVIJEK gubi novac! (Dutch Book)")

# Pokušaj protiv koherentnog
print("\nPOKUŠAJ PROTIV KOHERENTNOG AGENTA:")
print("="*36)
print("\nIste oklade s koherentnim agentom:")
print(f"  Oklada 3 sada: Agent prima {koherentan.vjerovanja['A_ili_B']*100:.0f}€ (ne 70€)")

print("\nMogući ishodi:")
for naziv, a_istina, b_istina in ishodi[::2]:  # Samo neki ishodi za kraćkću
    profit = 0
    detalji = []

    if a_istina:
        profit += 100 - 60
        detalji.append("+100")
    else:
        profit -= 60
        detalji.append("-60")

    if b_istina:
        profit += 100 - 50
        detalji.append("+100")
    else:
        profit -= 50
        detalji.append("-50")

    if a_istina or b_istina:
        profit += 80 - 100
        detalji.append("-100")
    else:
        profit += 0
        detalji.append("+0")

    print(f"  {naziv:8} {' '.join(detalji):15} = {profit:+.0f}€")

print("\nKoherentan agent može i dobiti i izgubiti.")
print("Nema sigurnog gubitka!")

print("\nEPISTEMOLOŠKA PORUKA:")
print("="*22)
print("\n• Nekoherentna vjerovanja vode u sigurni gubitak")
print("• Racionalnost zahtijeva poštivanje aksioma vjerojatnosti")
print("• Bayesov pristup garantira koherentnost")
print("• Pragmatičko opravdanje epistemologije!")
```

```
223 print("\n> \"Koherentnost nije sve, ali bez nje sve je ništa\" - de Finetti")
```

Output

```
Dutch Book Argument - Cijena nekoherentnosti
================================================

AGENT 1: Nekoherentan
Vjerovanja:
  P(A) = 0.60
  P(B) = 0.50
  P(A ∨ B) = 0.70 KRŠI AKSIOME!

  Trebalo bi biti: P(A ∨ B) ≥ max(0.60, 0.50) = 0.60
  i P(A ∨ B) ≤ 0.60 + 0.50 = 1.10 (ali ≤ 1)
  Koherentan raspon: [0.60, 1.00]

AGENT 2: Koherentan
Vjerovanja:
  P(A) = 0.60
  P(B) = 0.50
  P(A ∨ B) = 0.80  ✓ Zadovoljava aksiome

DUTCH BOOK PROTIV NEKOHERENTNOG AGENTA:
=======================================

Kladioničar konstruira sljedeće oklade:

Oklada 1: Agent se kladi ZA 'A'
  Fer cijena (prema agentu): 60.0€
  Dobitak ako A: 100€, Gubitak ako ¬A: -60€

Oklada 2: Agent se kladi ZA 'B'
  Fer cijena (prema agentu): 50.0€
  Dobitak ako B: 100€, Gubitak ako ¬B: -50€

Oklada 3: Agent se kladi PROTIV 'A ∨ B'
  Agent prima: 70.0€
  Mora platiti 100€ ako A ∨ B

ANALIZA ISHODA:
===============

Mogući ishodi:
  A∧B      +100 +100 -100  = +60€  → Agent dobiva 60€
  A∧¬B     +100 -50 -100   = -40€  → Agent gubi 40€
  ¬A∧B     -60 +100 -100   = -40€  → Agent gubi 40€
  ¬A∧¬B    -60 -50 +0      = -40€  → Agent gubi 40€

Agent UVIJEK gubi novac! (Dutch Book)
```

```
POKUŠAJ PROTIV KOHERENTNOG AGENTA:
==================================

Iste oklade s koherentnim agentom:
  Oklada 3 sada: Agent prima 80€ (ne 70€)

Mogući ishodi:
  A∧B      +100 +100 -100  = +70€
  ¬A∧B      -60 +100 -100   = -30€

Koherentan agent može i dobiti i izgubiti.
Nema sigurnog gubitka!

EPISTEMOLOŠKA PORUKA:
=====================

• Nekoherentna vjerovanja vode u sigurni gubitak
• Racionalnost zahtijeva poštivanje aksioma vjerojatnosti
• Bayesov pristup garantira koherentnost
• Pragmatičko opravdanje epistemologije!

> "Koherentnost nije sve, ali bez nje sve je ništa" - de Finetti
```

8.1.8 Prijedlozi za daljnje istraživanje

Za produbljivanje razumijevanja Bayesove epistemologije kroz praktične zadatke:

Goodmanov paradoks (grue/bleen)

Implementirajte problem "zeleno-plavih" smaragda. Pokažite kako različiti priori o prirodnim/neprirordnim predikatima utječu na induktivno zaključivanje.

Problem starih dokaza

Simulirajte situaciju gdje teorija objašnjava već poznate činjenice. Kako Bayesov pristup tretira dokaze koje znamo prije teorije?

Quine-Duhemova teza

Pokažite kako se negativan eksperiment može pripisati različitim hipotezama u teorijskoj mreži. Implementirajte Bayesovo ažuriranje za mrežu povezanih hipoteza.

Sleeping Beauty problem

Implementirajte ovaj paradoks samolociranja. Analizirajte sukob između "halfer" i "thirder" pozicija kroz Bayesov okvir.

Kuhnove znanstvene revolucije

Modelirajte promjenu paradigme kao nagli skok u posteriornim vjerojatnostima. Kada akumulirani dokazi dovode do "revolucije"?

Solomonoffova indukcija

Implementirajte univerzalnu indukciju koristeći Kolmogorovljevu složenost kao prior. Pokažite kako kraći opisi imaju veću apriornu vjerojatnost.

Akaike informacijski kriterij

Usporedite AIC s Bayesovim faktorom za selekciju modela. Pokažite trade-off između složenosti i točnosti.

Epistemička logika

Implementirajte modalne operatore znanja i vjerovanja. Pokažite kako se Bayesovo ažuriranje uklapa u formalnu epistemologiju.

Kauzalno zaključivanje

Implementirajte Pearl-ove do-operatore. Pokažite razliku između opažajne i intervencijske evidencije.

Socijalna epistemologija

Simulirajte mrežu agenata koji dijele dokaze. Istražite kako se konsenzus formira kroz Bayesovo ažuriranje.

Svaki zadatak istražuje dublje veze između vjerojatnosti, znanja i racionalnog zaključivanja.

8.1.9 Zaključak

Kroz ovu implementaciju istražili smo kako Bayesov teorem revolucionira naše razumijevanje znanja i znanosti:

1. **Znanje kao stupnjevito vjerovanje** - ne binarna istina
2. **Učenje kao ažuriranje** - ne zamjena vjerovanja
3. **Racionalnost kao koherentnost** - pragmatičko opravdanje
4. **Znanost kao konvergencija** - različiti putovi do istine

Bayesov pristup mijenja fundamentalna epistemološka pitanja:

> "Pitanje nije 'Što je istina?' već 'Koliko trebam vjerovati?'" - E.T. Jaynes

Ova promjena perspektive ima duboke implikacije:

- **Filozofija znanosti**: Od falsifikacije do stupnjeva potvrde
- **Epistemologija**: Od JTB (justified true belief) do racionalnog ažuriranja
- **Racionalnost**: Od logičke dedukcije do vjerojatnosnog zaključivanja

Kroz praktično programiranje otkrivamo da Bayesov teorem nije samo matematička formula, već **normativna teorija racionalnog vjerovanja** - pokazuje kako bi racionalni agent trebao mijenjati svoja uvjerenja suočen s dokazima.

> "Bayesov teorem je za neizvjesnost ono što je aritmetika za brojanje" - John Maynard Keynes

U svijetu nesavršenog znanja i nepotpunih informacija, Bayesova epistemologija nudi rigorozan okvir za navigiranje kroz neizvjesnost - ne obećava istinu, ali garantira racionalnost.

Poglavlje 9

Goodmanovi svijetovi: Problem indukcije i strojno učenje

9.1 Novi problem indukcije kroz prizmu računalnih znanosti

Nelson Goodman je 1955. godine formulirao **novi problem indukcije** koji pokazuje temeljnu poteškoću u razlikovanju valjanih od nevaljanih induktivnih zaključaka. Ovaj problem ima duboke implikacije za moderno strojno učenje.

> "Činjenica da se smaragd može jednako dobro opisati kao 'zelen' ili kao 'gruen' otkriva da svaki skup opažanja podržava beskonačno mnogo hipoteza." - Nelson Goodman

U ovoj bilježnici istražujemo kako se Goodmanova zagadka manifestira u kontekstu strojnog učenja kroz **No-Free-Lunch teoreme**.

9.1.1 Klasični problem indukcije

Prije nego što uronimo u Goodmanov novi problem, razmotrimo klasični **Humeov problem indukcije**:

$$\text{Opaženi slučajevi} \xrightarrow{?} \text{Opći zaključak}$$

Induktivno zaključivanje pokušava iz konačnog broja opažanja izvesti općeniti zakon. To je temelj znanosti, ali filozofski gledano - nema logičke nužnosti da će se buduća opažanja ponašati kao prošla.

```
1 from dataclasses import dataclass
```

```python
from datetime import datetime, timedelta
import random

@dataclass
class Opažanje:
    """Predstavlja jedno empirijsko opažanje."""
    objekt: str
    svojstvo: str
    vrijeme: datetime

    def __repr__(self):
        return f"{self.objekt}: {self.svojstvo} (vrijeme: {self.vrijeme.date()})"

# Generiraj opažanja smaragda
def generiraj_opažanja(n=5, svojstvo="zelen"):
    """Generira niz opažanja smaragda."""
    opažanja = []
    početak = datetime(2020, 1, 1)

    for i in range(n):
        vrijeme = početak + timedelta(days=random.randint(i*200, (i+1)*200))
        opažanja.append(Opažanje(f"Smaragd {i+1}", svojstvo, vrijeme))

    return opažanja

# Klasična indukcija
opažanja_zeleni = generiraj_opažanja(5, "zelen")

print("Klasična indukcija:")
print("="*20)
print("Opažanja smaragda:")
for op in opažanja_zeleni:
    print(f"  {op}")
print("\nInduktivni zaključak: Svi smaragdi su zeleni")
```

Output

```
Klasična indukcija:
====================
Opažanja smaragda:
  Smaragd 1: zelen (vrijeme: 2020-03-17)
  Smaragd 2: zelen (vrijeme: 2021-01-21)
  Smaragd 3: zelen (vrijeme: 2021-06-04)
  Smaragd 4: zelen (vrijeme: 2022-03-01)
  Smaragd 5: zelen (vrijeme: 2022-07-22)

Induktivni zaključak: Svi smaragdi su zeleni
```

9.1.2 Goodmanov "grue" predikat

Goodman uvodi novi predikat **"grue"** (kombinacija "green" i "blue"):

$$\text{grue}(x) = \begin{cases} \text{zelen}(x) & \text{ako je } x \text{ opažen prije } t_0 \\ \text{plav}(x) & \text{ako je } x \text{ opažen nakon } t_0 \end{cases}$$

gdje je t_0 neki budući trenutak (npr. 1. siječnja 2100.).

Paradoks: Sva dosadašnja opažanja zelenih smaragda jednako dobro potvrđuju hipotezu "svi smaragdi su zeleni" kao i hipotezu "svi smaragdi su grueni"!

```python
class GruePredikat:

    """Implementacija Goodmanovog 'grue' predikata."""

    def __init__(self, kritični_trenutak):
        self.t0 = kritični_trenutak

    def je_grue(self, objekt, vrijeme_opažanja):
        """Provjerava je li objekt 'grue' u danom trenutku."""
        if vrijeme_opažanja < self.t0:
            # Prije t0: grue = zelen
            return objekt.svojstvo == "zelen"
        else:
            # Nakon t0: grue = plav
            return objekt.svojstvo == "plav"

    def predviđa(self, vrijeme):
        """Što predviđa grue hipoteza za dano vrijeme."""
        if vrijeme < self.t0:
            return "zelen"
        else:
            return "plav"

# Definiraj kritični trenutak
t0 = datetime(2100, 1, 1)
grue_predikat = GruePredikat(t0)

print("Goodmanov problem indukcije:")
print("="*29)
print(f"Kritični trenutak t₀: {t0.date()}")
print("\nProvjera hipoteza za prošla opažanja:")
print("-"*38)

# Testiraj obje hipoteze na istim podacima
for op in opažanja_zeleni[:3]:  # Prikaži prva 3
    print(f"{op.objekt} ({op.vrijeme.date()}):")
    print(f"  Opaženo: {op.svojstvo}")
```

```
38
39      # Hipoteza 1: Svi smaragdi su zeleni
40      h1_istina = op.svojstvo == "zelen"
41      print(f"  H1 (zeleni): {'⊤' if h1_istina else '⊥'} (predviđa: zelen)")
42
43      # Hipoteza 2: Svi smaragdi su grueni
44      h2_predviđanje = grue_predikat.predviđa(op.vrijeme)
45      h2_istina = op.svojstvo == h2_predviđanje
46      print(f"  H2 (grueni): {'⊤' if h2_istina else '⊥'} (predviđa: {h2_predviđanje} -
        ↪ {'prije' if op.vrijeme < t0 else 'nakon'} t₀)")
47      print()
48
49  print("Rezultat: Obje hipoteze jednako dobro objašnjavaju sva opažanja!")
50  print(f"\nPredviđanja za budućnost (nakon {t0.date()}):")
51  print("-"*46)
52  print("  H1 (svi zeleni): smaragdi će biti zeleni")
53  print("  H2 (svi grueni): smaragdi će biti plavi")
```

Output

```
Goodmanov problem indukcije:
============================
Kritični trenutak t₀: 2100-01-01

Provjera hipoteza za prošla opažanja:
--------------------------------------
Smaragd 1 (2020-03-17):
  Opaženo: zelen
  H1 (zeleni): ⊤ (predviđa: zelen)
  H2 (grueni): ⊤ (predviđa: zelen - prije t₀)

Smaragd 2 (2021-01-21):
  Opaženo: zelen
  H1 (zeleni): ⊤ (predviđa: zelen)
  H2 (grueni): ⊤ (predviđa: zelen - prije t₀)

Smaragd 3 (2021-06-04):
  Opaženo: zelen
  H1 (zeleni): ⊤ (predviđa: zelen)
  H2 (grueni): ⊤ (predviđa: zelen - prije t₀)

Rezultat: Obje hipoteze jednako dobro objašnjavaju sva opažanja!

Predviđanja za budućnost (nakon 2100-01-01):
----------------------------------------------
  H1 (svi zeleni): smaragdi će biti zeleni
  H2 (svi grueni): smaragdi će biti plavi
```

9.1.3 Beskonačnost alternativnih hipoteza

Goodmanov argument pokazuje da za svaki skup opažanja postoji **beskonačno mnogo** jednako dobro podržanih hipoteza. Možemo konstruirati "grue$_1$", "grue$_2$", ... sa različitim kritičnim trenucima:

$$\text{grue}_n(x) = \begin{cases} \text{zelen}(x) & \text{ako je } x \text{ opažen prije } t_n \\ \text{plav}(x) & \text{ako je } x \text{ opažen nakon } t_n \end{cases}$$

```python
def stvori_grue_hipotezu(kritični_trenutak, naziv):
    """Stvara grue hipotezu s danim kritičnim trenutkom."""
    class GrueHipoteza:
        def __init__(self):
            self.t0 = kritični_trenutak
            self.naziv = naziv

        def predviđa(self, vrijeme):
            return "zelen" if vrijeme < self.t0 else "plav"

        def provjeri(self, opažanje):
            predviđanje = self.predviđa(opažanje.vrijeme)
            return opažanje.svojstvo == predviđanje

    return GrueHipoteza()

# Stvori više alternativnih hipoteza
hipoteze = {
    "standard": type('StandardHipoteza', (), {
        'naziv': 'standard',
        'predviđa': lambda self, t: "zelen",
        'provjeri': lambda self, op: op.svojstvo == "zelen"
    })(),
    "grue_2030": stvori_grue_hipotezu(datetime(2030, 1, 1), "grue_2030"),
    "grue_2050": stvori_grue_hipotezu(datetime(2050, 1, 1), "grue_2050"),
    "grue_2100": stvori_grue_hipotezu(datetime(2100, 1, 1), "grue_2100"),
    "grue_2200": stvori_grue_hipotezu(datetime(2200, 1, 1), "grue_2200"),
}

print("Beskonačnost alternativnih hipoteza:")
print("="*37)
print("\nZa iste podatke možemo konstruirati mnoštvo hipoteza:\n")

# Prikaži predviđanja svake hipoteze
test_vremena = [datetime(2050, 6, 15), datetime(2150, 6, 15)]

for naziv, hipoteza in hipoteze.items():
    if naziv == "standard":
        print(f"Hipoteza '{naziv}': Svi smaragdi su uvijek zeleni")
    else:
        t0_godina = int(naziv.split('_')[1])
```

```
        print(f"Hipoteza '{naziv}': grueni s prijelazom {t0_godina}-01-01")

    for t in test_vremena:
        print(f"  Predviđanje za {t.year}: {hipoteza.predviđa(t)}")
    print()

# Provjeri konzistentnost s postojećim opažanjima
print("Provjera konzistentnosti s opažanjima:")
print("-"*40)

for naziv, hipoteza in hipoteze.items():
    rezultati = [hipoteza.provjeri(op) for op in opažanja_zeleni]
    simboli = "".join(['⊤' if r else '⊥' for r in rezultati])
    točnih = sum(rezultati)
    print(f"{naziv}: {simboli} ({točnih}/{len(rezultati)} točnih)")

print("\nSve hipoteze su jednako dobro podržane postojećim podacima!")
```

Output

```
Beskonačnost alternativnih hipoteza:
====================================

Za iste podatke možemo konstruirati mnoštvo hipoteza:

Hipoteza 'standard': Svi smaragdi su uvijek zeleni
  Predviđanje za 2050: zelen
  Predviđanje za 2150: zelen

Hipoteza 'grue_2030': grueni s prijelazom 2030-01-01
  Predviđanje za 2050: plav
  Predviđanje za 2150: plav

Hipoteza 'grue_2050': grueni s prijelazom 2050-01-01
  Predviđanje za 2050: plav
  Predviđanje za 2150: plav

Hipoteza 'grue_2100': grueni s prijelazom 2100-01-01
  Predviđanje za 2050: zelen
  Predviđanje za 2150: plav

Hipoteza 'grue_2200': grueni s prijelazom 2200-01-01
  Predviđanje za 2050: zelen
  Predviđanje za 2150: zelen

Provjera konzistentnosti s opažanjima:
----------------------------------------
standard: ⊤⊤⊤⊤⊤ (5/5 točnih)
grue_2030: ⊤⊤⊤⊤⊤ (5/5 točnih)
grue_2050: ⊤⊤⊤⊤⊤ (5/5 točnih)
```

```
grue_2100: TTTTT (5/5 točnih)
grue_2200: TTTTT (5/5 točnih)

Sve hipoteze su jednako dobro podržane postojećim podacima!
```

9.1.4 No-Free-Lunch teoremi u strojnom učenju

No-Free-Lunch (NFL) teoremi pokazuju da je Goodmanov problem duboko ukorijenjen u strojnom učenju. Teorem kaže:

$$\sum_{f \in \mathcal{F}} \mathrm{GP}(L, f) = 0.5$$

gdje je:

- L - bilo koji algoritam učenja
- $\mathcal{F}$ - skup svih mogućih ciljnih funkcija
- GP - generalizacijska performansa

Značenje: Prosječna performansa bilo kojeg algoritma učenja preko svih mogućih problema jednaka je slučajnom pogađanju!

```python
import itertools

class BooleanConcept:
    """Predstavlja Booleovu funkciju kao koncept za učenje."""

    def __init__(self, truth_table, naziv=""):
        self.tablica = truth_table
        self.naziv = naziv

    def evaluiraj(self, ulaz):
        """Evaluira funkciju za dani ulaz."""
        return self.tablica.get(ulaz, False)

    def konzistentan_s(self, skup_učenja):
        """Provjerava je li koncept konzistentan sa skupom za učenje."""
        for ulaz, izlaz in skup_učenja.items():
            if self.evaluiraj(ulaz) != izlaz:
                return False
        return True

# Definiraj skup za učenje (parcijalna tablica istinitosti)
skup_učenja = {
```

```
    (False, False): False,
    (False, True): True,
    (True, False): True,
    # (True, True) nije u skupu za učenje!
}

# Konstruiraj dva različita koncepta konzistentna s podacima
# Koncept 1: XOR
xor_tablica = {
    (False, False): False,
    (False, True): True,
    (True, False): True,
    (True, True): False  # XOR
}

# Koncept 2: OR
or_tablica = {
    (False, False): False,
    (False, True): True,
    (True, False): True,
    (True, True): True  # OR
}

koncepti = [
    BooleanConcept(xor_tablica, "XOR funkcija"),
    BooleanConcept(or_tablica, "OR funkcija")
]

print("No-Free-Lunch demonstracija:")
print("="*29)
print()

# Prikaži skup za učenje
print("Skup za učenje: ", end="")
print("{", end="")
for i, (ulaz, izlaz) in enumerate(skup_učenja.items()):
    if i > 0:
        print(", ", end="")
    ulaz_str = f"({'⊤' if ulaz[0] else '⊥'}, {'⊤' if ulaz[1] else '⊥'})"
    izlaz_str = '⊤' if izlaz else '⊥'
    print(f"{ulaz_str}: {izlaz_str}", end="")
print("}")

testni = (True, True)
print(f"Testni primjer: ({'⊤' if testni[0] else '⊥'}, {'⊤' if testni[1] else '⊥'}) = ?")
print()

print("Mogući koncepti konzistentni s podacima:")
print("-"*42)

for i, koncept in enumerate(koncepti, 1):
    print(f"Koncept {i}: {koncept.naziv}")

```

```
    # Predviđanje za testni primjer
    pred = koncept.evaluiraj(testni)
    print(f"  Predviđanje za ({'⊤' if testni[0] else '⊥'}, {'⊤' if testni[1] else '⊥'}):
    ↪ {'⊤' if pred else '⊥'}")

    # Prikaži cijelu tablicu
    print("  Tablica:")
    for ulaz in [(False, False), (False, True), (True, False), (True, True)]:
        izlaz = koncept.evaluiraj(ulaz)
        ulaz_str = f"({'⊤' if ulaz[0] else '⊥'}, {'⊤' if ulaz[1] else '⊥'})"
        izlaz_str = '⊤' if izlaz else '⊥'
        oznaka = " ✓" if ulaz in skup_učenja else ""
        print(f"    {ulaz_str} → {izlaz_str}{oznaka}")
    print()

print("Oba koncepta su jednako valjana za dane podatke!")
print("Ali daju različita predviđanja za neviđene primjere.")
```

Output

```
No-Free-Lunch demonstracija:
============================

Skup za učenje: {(⊥, ⊥): ⊥, (⊥, ⊤): ⊤, (⊤, ⊥): ⊤}
Testni primjer: (⊤, ⊤) = ?

Mogući koncepti konzistentni s podacima:
----------------------------------------
Koncept 1: XOR funkcija
  Predviđanje za (⊤, ⊤): ⊥
  Tablica:
    (⊥, ⊥) → ⊥ ✓
    (⊥, ⊤) → ⊤ ✓
    (⊤, ⊥) → ⊤ ✓
    (⊤, ⊤) → ⊥

Koncept 2: OR funkcija
  Predviđanje za (⊤, ⊤): ⊤
  Tablica:
    (⊥, ⊥) → ⊥ ✓
    (⊥, ⊤) → ⊤ ✓
    (⊤, ⊥) → ⊤ ✓
    (⊤, ⊤) → ⊤

Oba koncepta su jednako valjana za dane podatke!
Ali daju različita predviđanja za neviđene primjere.
```

9.1.5 Konstrukcija "savijenih" koncepata

NFL teoremi pokazuju da za svaki koncept C koji algoritam učenja dobro nauči, postoji "savijeni" koncept C' koji će biti loše naučen:

$$C'(x) = \begin{cases} C(x) & \text{ako } x \in \text{SkupUčenja} \\ \neg C(x) & \text{ako } x \notin \text{SkupUčenja} \end{cases}$$

Ovo je direktna analogija s Goodmanovim "grue" predikatom!

```python
import numpy as np

def stvori_savijeni_koncept(originalni_koncept, skup_učenja):
    """Stvara 'savijeni' koncept koji se slaže na skupu učenja ali ne generalizira."""

    class SavijeniKoncept:
        def __init__(self):
            self.originalni = originalni_koncept
            self.memorija = skup_učenja

        def predviđa(self, primjer):
            # Ako je primjer u skupu učenja, koristi originalnu funkciju
            for x_mem, _ in self.memorija:
                if np.allclose(primjer, x_mem):
                    return self.originalni(primjer)

            # Inače, vrati suprotno od originalne funkcije
            return not self.originalni(primjer)

    return SavijeniKoncept()

# Definiraj jednostavan linearni koncept
def linearni_koncept(x):
    """Jednostavan linearni klasifikator: x[0] + x[1] > 1"""
    return x[0] + x[1] > 1

# Generiraj skup za učenje
np.random.seed(42)
skup_učenja_ml = []
for _ in range(6):
    x = np.random.rand(2)
    y = linearni_koncept(x)
    skup_učenja_ml.append((x, y))

# Stvori savijeni koncept
savijeni = stvori_savijeni_koncept(linearni_koncept, skup_učenja_ml)

print("Konstrukcija 'savijenih' koncepata:")
print("="*36)
```

```
print("\nOriginalni koncept C: jednostavna linearna granica")
print(f"Skup za učenje: {len(skup_učenja_ml)} primjera")
print()

# Testiraj na skupu za učenje
print("Performanse na skupu za učenje:")
print("-"*32)
točnih_C = 0
točnih_C_prime = 0

for x, y in skup_učenja_ml:
    pred_C = linearni_koncept(x)
    pred_C_prime = savijeni.predviđa(x)

    print(f"Primjer ({x[0]:.1f}, {x[1]:.1f}) → ", end="")
    print(f"Očekivano: {'⊤' if y else '⊥'}, ", end="")
    print(f"C: {'⊤' if pred_C else '⊥'} {'✓' if pred_C == y else '×'}, ", end="")
    print(f"C': {'⊤' if pred_C_prime else '⊥'} {'✓' if pred_C_prime == y else '×'}")

    if pred_C == y:
        točnih_C += 1
    if pred_C_prime == y:
        točnih_C_prime += 1

print(f"\nTočnost na skupu za učenje:")
print(f"  Koncept C: {100 * točnih_C / len(skup_učenja_ml):.1f}%")
print(f"  Koncept C': {100 * točnih_C_prime / len(skup_učenja_ml):.1f}%")

# Testiraj na novim primjerima
print("\nPerformanse na testnom skupu:")
print("-"*30)

testni_skup = [np.random.rand(2) for _ in range(5)]
točnih_test_C = 0
točnih_test_C_prime = 0

for x_test in testni_skup:
    y_pravi = linearni_koncept(x_test)  # "Prava" oznaka
    pred_C = linearni_koncept(x_test)
    pred_C_prime = savijeni.predviđa(x_test)

    print(f"Test ({x_test[0]:.1f}, {x_test[1]:.1f}) → ", end="")
    print(f"C: {'⊤' if pred_C else '⊥'}, ", end="")
    print(f"C': {'⊤' if pred_C_prime else '⊥'}", end="")
    if pred_C != pred_C_prime:
        print(" (različito!)")
    else:
        print()

    if pred_C == y_pravi:
        točnih_test_C += 1
    if pred_C_prime == y_pravi:
        točnih_test_C_prime += 1
```

```
93
94 print(f"\nTočnost na testnom skupu:")
95 print(f"  Koncept C: {100 * točnih_test_C / len(testni_skup):.1f}% (dobar)")
96 print(f"  Koncept C': {100 * točnih_test_C_prime / len(testni_skup):.1f}% (loš!)")
97 print("\nC' je 'savijen' - slaže se s C na skupu za učenje,")
98 print("ali se ponaša suprotno na novim primjerima!")
```

Output

```
Konstrukcija 'savijenih' koncepata:
====================================

Originalni koncept C: jednostavna linearna granica
Skup za učenje: 6 primjera

Performanse na skupu za učenje:
--------------------------------
Primjer (0.4, 1.0) → Očekivano: ⊤, C: ⊤ ✓, C': ⊤ ✓
Primjer (0.7, 0.6) → Očekivano: ⊤, C: ⊤ ✓, C': ⊤ ✓
Primjer (0.2, 0.2) → Očekivano: ⊥, C: ⊥ ✓, C': ⊥ ✓
Primjer (0.1, 0.9) → Očekivano: ⊥, C: ⊥ ✓, C': ⊥ ✓
Primjer (0.6, 0.7) → Očekivano: ⊤, C: ⊤ ✓, C': ⊤ ✓
Primjer (0.0, 1.0) → Očekivano: ⊥, C: ⊥ ✓, C': ⊥ ✓

Točnost na skupu za učenje:
  Koncept C: 100.0%
  Koncept C': 100.0%

Performanse na testnom skupu:
------------------------------
Test (0.8, 0.2) → C: ⊤, C': ⊥ (različito!)
Test (0.2, 0.2) → C: ⊥, C': ⊤ (različito!)
Test (0.3, 0.5) → C: ⊥, C': ⊤ (različito!)
Test (0.4, 0.3) → C: ⊥, C': ⊤ (različito!)
Test (0.6, 0.1) → C: ⊥, C': ⊤ (različito!)

Točnost na testnom skupu:
  Koncept C: 100.0% (dobar)
  Koncept C': 0.0% (loš!)

C' je 'savijen' - slaže se s C na skupu za učenje,
ali se ponaša suprotno na novim primjerima!
```

9.1.6 Implikacije za strojno učenje

Induktivni bias kao nužnost

NFL teoremi i Goodmanov problem pokazuju da **induktivni bias nije nedostatak već nužnost** svakog sustava učenja. Bez pretpostavki o prirodi problema, učenje je nemoguće.

Različiti algoritmi imaju različite induktivne pristranosti:

- **Linearna regresija**: pretpostavlja linearnu vezu
- **Stabla odlučivanja**: pretpostavljaju hijerarhijsku strukturu
- **Neuronske mreže**: pretpostavljaju kompozicionalnost
- **k-NN**: pretpostavlja lokalnu glatkoću

```python
def najbliži_susjed(točka, skup_učenja):
    """Jednostavan 1-NN klasifikator."""
    min_udaljenost = float('inf')
    najbliža_oznaka = None

    for x, y in skup_učenja:
        udaljenost = np.sqrt((točka[0] - x[0])**2 + (točka[1] - x[1])**2)
        if udaljenost < min_udaljenost:
            min_udaljenost = udaljenost
            najbliža_oznaka = y

    return najbliža_oznaka

def xor_funkcija(x):
    """XOR funkcija: istinito ako je točno jedan ulaz > 0.5."""
    return (x[0] > 0.5) != (x[1] > 0.5)

# Generiraj XOR podatke
xor_podaci = [
    (np.array([0.1, 0.1]), False),
    (np.array([0.1, 0.9]), True),
    (np.array([0.9, 0.1]), True),
    (np.array([0.9, 0.9]), False),
    (np.array([0.3, 0.3]), False),
    (np.array([0.3, 0.7]), True),
    (np.array([0.7, 0.3]), True),
    (np.array([0.7, 0.7]), False),
]

# Definiraj različite algoritme s različitim biasima
algoritmi = [
    ("Linearna granica (bias: linearnost)",
     lambda x: x[0] + x[1] > 1),
```

```
    ("Najbliži susjed (bias: lokalna glatkoća)",
     lambda x: najbliži_susjed(x, xor_podaci)),
    ("XOR funkcija (bias: točno odgovara problemu)",
     xor_funkcija),
    ("Uvijek ⊤ (bias: konstantnost)",
     lambda x: True),
]

print("Utjecaj induktivnog biasa:")
print("="*27)
print(f"\nPodatkovni skup: {len(xor_podaci)} točaka koje čine XOR uzorak")
print("\nRazličiti algoritmi s različitim biasima:")
print("-"*43)

test_točke = [np.array([0.5, 0.5]), np.array([0.2, 0.8])]

for i, (naziv, algoritam) in enumerate(algoritmi, 1):
    print(f"\n{i}. {naziv}")

    # Testiraj na nekoliko točaka
    for točka in test_točke:
        pred = algoritam(točka)
        print(f"   Predviđanje za ({točka[0]:.1f}, {točka[1]:.1f}): {'⊤' if pred else '⊥'}")

    # Izračunaj točnost
    točnih = sum(1 for x, y in xor_podaci if algoritam(x) == y)
    točnost = 100 * točnih / len(xor_podaci)
    print(f"   Točnost na XOR podacima: {točnost:.1f}%")

    # Komentar
    if točnost > 90:
        print("   → Savršen jer ima pravi bias")
    elif točnost > 60:
        print("   → Bolji, ali još uvijek ograničen")
    else:
        if "linearnost" in naziv:
            print("   → Loš za XOR zbog krivog biasa")
        else:
            print("   → Loš zbog previše jednostavnog biasa")

print("\nZaključak: Uspjeh učenja ovisi o podudaranju")
print("između induktivnog biasa i stvarnog problema!")
```

Output

```
Utjecaj induktivnog biasa:
===========================

Podatkovni skup: 8 točaka koje čine XOR uzorak

Različiti algoritmi s različitim biasima:
```

```
----------------------------------------

1. Linearna granica (bias: linearnost)
   Predviđanje za (0.5, 0.5): ⊥
   Predviđanje za (0.2, 0.8): ⊥
   Točnost na XOR podacima: 25.0%
   → Loš za XOR zbog krivog biasa

2. Najbliži susjed (bias: lokalna glatkoća)
   Predviđanje za (0.5, 0.5): ⊥
   Predviđanje za (0.2, 0.8): ⊤
   Točnost na XOR podacima: 100.0%
   → Savršen jer ima pravi bias

3. XOR funkcija (bias: točno odgovara problemu)
   Predviđanje za (0.5, 0.5): ⊥
   Predviđanje za (0.2, 0.8): ⊤
   Točnost na XOR podacima: 100.0%
   → Savršen jer ima pravi bias

4. Uvijek ⊤ (bias: konstantnost)
   Predviđanje za (0.5, 0.5): ⊤
   Predviđanje za (0.2, 0.8): ⊤
   Točnost na XOR podacima: 50.0%
   → Loš zbog previše jednostavnog biasa

Zaključak: Uspjeh učenja ovisi o podudaranju
između induktivnog biasa i stvarnog problema!
```

9.1.7 Filozofske implikacije

Projektibilnost vs. neprojektibilnost

Goodman razlikuje **projektibilne** i **neprojektibilne** predikate:

- **Projektibilni**: "zelen", "okrugao", "teži od 1kg"
- **Neprojektibilni**: "grue", "gruen s prijelazom 2100."

Ali što čini predikat projektibilnim? Goodman sugerira da je to stvar **ukorijenjenosti** (*entrenchment*) u našem jeziku i praksi.

Implikacije za AI i AGI

1. **Nema univerzalnog algoritma učenja** - svaki mora imati bias
2. **Prijelaz od podataka na znanje** zahtijeva pretpostavke

3. **Ljudska inteligencija** možda uspijeva jer ima evolucijski oblikovane biase
4. **AGI sustavi** moraju riješiti problem izbora pravog biasa

```python
import random

class Agent:
    """Agent s određenim induktivnim biasom."""

    def __init__(self, bias_tip):
        self.bias = bias_tip
        self.fitness = 0

    def predviđa(self, x):
        """Predviđanje ovisno o biasu."""
        if self.bias == "linearni":
            return x[0] + x[1] > 1
        elif self.bias == "kvadratni":
            return x[0]**2 + x[1]**2 > 0.5
        elif self.bias == "neutralni":
            return random.choice([True, False])
        elif self.bias == "složeni_1":
            return (x[0] > 0.5) != (x[1] > 0.5)  # Približno XOR
        elif self.bias == "složeni_2":
            return abs(x[0] - x[1]) > 0.3
        elif self.bias == "kvadratni_modificiran":
            return (x[0] - 0.5)**2 + (x[1] - 0.5)**2 < 0.3
        else:
            return False

    def evaluiraj(self, test_podaci):
        """Evaluira agenta na test podacima."""
        točnih = 0
        for x, y in test_podaci:
            if self.predviđa(x) == y:
                točnih += 1
        self.fitness = točnih / len(test_podaci)
        return self.fitness

def evolucija_biasa(generacije=20, veličina_populacije=20):
    """Simulira evoluciju induktivnog biasa."""

    # Mogući biasi
    biasi = ["linearni", "kvadratni", "neutralni",
             "složeni_1", "složeni_2", "kvadratni_modificiran"]

    # Stvori početnu populaciju
    populacija = [Agent(random.choice(biasi)) for _ in range(veličina_populacije)]

    # Test podaci (XOR problem)
    test_podaci = [
        (np.array([0.2, 0.2]), False),
```

```
        (np.array([0.2, 0.8]), True),
        (np.array([0.8, 0.2]), True),
        (np.array([0.8, 0.8]), False),
        (np.array([0.5, 0.1]), False),
        (np.array([0.1, 0.5]), False),
        (np.array([0.9, 0.5]), True),
        (np.array([0.5, 0.9]), True),
        (np.array([0.4, 0.4]), False),
        (np.array([0.6, 0.6]), False),
        (np.array([0.3, 0.7]), True),
        (np.array([0.7, 0.3]), True),
    ]

    print("Simulacija evolucije induktivnog biasa:")
    print("="*41)
    print(f"\nInicijalna populacija: {veličina_populacije} agenata s različitim biasima")
    print("Okoliš: jednostavan XOR svijet")
    print("\nEvolucija kroz generacije:")
    print("-"*27)

    povijest = []

    for gen in range(generacije):
        # Evaluiraj sve agente
        for agent in populacija:
            agent.evaluiraj(test_podaci)

        # Sortiraj po fitness-u
        populacija.sort(key=lambda a: a.fitness, reverse=True)

        # Zapisivanje najboljih
        if gen % 5 == 0 or gen == generacije - 1:
            najbolji = populacija[0]
            print(f"Generacija {gen+1}: Najbolji bias = {najbolji.bias} "
                  f"({najbolji.fitness*100:.1f}% točnosti)")

        # Selekcija i reprodukcija (elitizam + turnir)
        nova_populacija = populacija[:5]  # Zadrži najboljih 5

        while len(nova_populacija) < veličina_populacije:
            # Turnirska selekcija
            turnir = random.sample(populacija[:10], 2)
            pobjednik = max(turnir, key=lambda a: a.fitness)

            # Stvori novog agenta s mogućom mutacijom
            if random.random() < 0.1:  # 10% šanse za mutaciju
                novi_bias = random.choice(biasi)
            else:
                novi_bias = pobjednik.bias

            nova_populacija.append(Agent(novi_bias))

        populacija = nova_populacija
```

```
102
103     # Finalna statistika
104     print("\nDistribucija biasa u finaliznoj populaciji:")
105     print("-"*45)
106
107     bias_count = {}
108     for agent in populacija:
109         bias_count[agent.bias] = bias_count.get(agent.bias, 0) + 1
110
111     for bias, count in sorted(bias_count.items(), key=lambda x: x[1], reverse=True):
112         postotak = 100 * count / veličina_populacije
113         print(f"{bias}: {postotak:.1f}%")
114
115     print("\nZaključak: Evolucija prirodno selektira biase")
116     print("koji odgovaraju strukturi okoliša!")
117
118 # Pokreni simulaciju
119 evolucija_biasa()
```

Output

```
Simulacija evolucije induktivnog biasa:
========================================

Inicijalna populacija: 20 agenata s različitim biasima
Okoliš: jednostavan XOR svijet

Evolucija kroz generacije:
--------------------------
Generacija 1: Najbolji bias = kvadratni (83.3% točnosti)
Generacija 6: Najbolji bias = složeni_1 (100.0% točnosti)
Generacija 11: Najbolji bias = složeni_1 (100.0% točnosti)
Generacija 16: Najbolji bias = složeni_1 (100.0% točnosti)
Generacija 20: Najbolji bias = složeni_1 (100.0% točnosti)

Distribucija biasa u finaliznoj populaciji:
--------------------------------------------
složeni_1: 95.0%
složeni_2: 5.0%

Zaključak: Evolucija prirodno selektira biase
koji odgovaraju strukturi okoliša!
```

9.1.8 Praktične implikacije i rješenja

Kako se nositi s Goodmanovim problemom u praksi?

1. **Regularizacija** - ograničava složenost hipoteza

2. **Križna validacija** - testira generalizaciju
3. **Occamova oštrica** - preferira jednostavnije hipoteze
4. **Domensko znanje** - koristi ljudsko znanje o problemu
5. **Ansambl metode** - kombinira različite biase

```python
def occamova_oštrica(hipoteze):
    """Odabire najjednostavniju hipotezu."""
    # Definiraj složenost kao broj parametara/prijelaza
    složenosti = {}
    for naziv, hip in hipoteze.items():
        if naziv == "standard":
            složenosti[naziv] = 1  # Najjednostavnija
        else:
            # Grue hipoteze imaju dodatni parametar (vrijeme prijelaza)
            složenosti[naziv] = 2

    # Odaberi hipotezu s najmanjom složenošću
    najjednostavnija = min(složenosti.items(), key=lambda x: x[1])
    return najjednostavnija[0], složenosti

def bayesov_pristup(hipoteze, opažanja, priori):
    """Koristi Bayesovo zaključivanje s priorima."""
    posteriori = {}

    for naziv, hip in hipoteze.items():
        # Likelihood - sve hipoteze objašnjavaju podatke jednako dobro
        likelihood = 1.0  # Pojednostavljeno

        # Posterior proporcionalan je prior * likelihood
        posteriori[naziv] = priori[naziv] * likelihood

    # Normaliziraj
    ukupno = sum(posteriori.values())
    for naziv in posteriori:
        posteriori[naziv] /= ukupno

    # Odaberi hipotezu s najvećim posteriorom
    najbolja = max(posteriori.items(), key=lambda x: x[1])
    return najbolja[0], posteriori

def ansambl_metoda(hipoteze, težine):
    """Kombinira predviđanja više hipoteza."""
    test_vremena = [
        datetime(2025, 1, 1),
        datetime(2040, 1, 1),
        datetime(2060, 1, 1),
        datetime(2110, 1, 1)
    ]

```

```
    predviđanja = {}
    for t in test_vremena:
        glasovi = {"zelen": 0, "plav": 0}

        for naziv, hip in hipoteze.items():
            pred = hip.predviđa(t)
            glasovi[pred] += težine[naziv]

        predviđanja[t.year] = glasovi

    return predviđanja

# Demonstracija
print("Praktična rješenja za Goodmanov problem:")
print("="*41)
print("\nTest različitih pristupa na 'grue' problemu:")

# Pripremi hipoteze
test_hipoteze = {
    "standard": hipoteze["standard"],
    "grue_2030": hipoteze["grue_2030"],
    "grue_2050": hipoteze["grue_2050"],
    "grue_2100": hipoteze["grue_2100"]
}

# 1. Bez regularizacije
print("\n1. Bez regularizacije (prihvaća sve hipoteze):")
print(f"   Razmatrane hipoteze: {', '.join(test_hipoteze.keys())}")
print("   Odabrana: standard (proizvoljan izbor)")
print("   Problem: Nema kriterija za izbor!")

# 2. Occamova oštrica
print("\n2. Occamova oštrica (preferira jednostavnije):")
najbolja_occam, složenosti = occamova_oštrica(test_hipoteze)
print("   Složenost hipoteza:")
for naziv, slož in sorted(složenosti.items(), key=lambda x: x[1]):
    komentar = " (najjednostavnija)" if slož == 1 else ""
    print(f"     {naziv}: {slož}{komentar}")
print(f"   Odabrana: {najbolja_occam}")
print("   Razlog: Najniža složenost")

# 3. Bayesov pristup
print("\n3. Bayesov pristup (koristi priore):")
priori = {
    "standard": 0.7,  # Visok prior za standardnu hipotezu
    "grue_2030": 0.1,
    "grue_2050": 0.1,
    "grue_2100": 0.1
}
najbolja_bayes, posteriori = bayesov_pristup(test_hipoteze, opažanja_zeleni, priori)

print("   Prior vjerojatnosti:")
for naziv, p in priori.items():
```

```
    print(f"     {naziv}: {p:.3f}")
print("   Posterior (nakon opažanja):")
for naziv, p in posteriori.items():
    print(f"     {naziv}: {p:.3f}")
print(f"   Odabrana: {najbolja_bayes}")
print("   Razlog: Najviši posterior")

# 4. Ansambl metoda
print("\n4. Ansambl metoda (kombinira hipoteze):")
težine = {
    "standard": 0.7,
    "grue_2030": 0.1,
    "grue_2050": 0.1,
    "grue_2100": 0.1
}
ansambl_pred = ansambl_metoda(test_hipoteze, težine)
print("   Težinski prosjek predviđanja:")
for godina, glasovi in ansambl_pred.items():
    ukupno = sum(glasovi.values())
    postotak_zelen = 100 * glasovi["zelen"] / ukupno
    postotak_plav = 100 * glasovi["plav"] / ukupno
    print(f"   Za {godina}: {postotak_zelen:.1f}% zeleno, {postotak_plav:.1f}% plavo")
print("   Predviđanje: Ponderirana kombinacija")

print("\nZaključak: Praktična rješenja koriste dodatne")
print("kriterije izvan čiste logike za izbor hipoteza.")
```

Output

```
Praktična rješenja za Goodmanov problem:
========================================

Test različitih pristupa na 'grue' problemu:

1. Bez regularizacije (prihvaća sve hipoteze):
   Razmatrane hipoteze: standard, grue_2030, grue_2050, grue_2100
   Odabrana: standard (proizvoljan izbor)
   Problem: Nema kriterija za izbor!

2. Occamova oštrica (preferira jednostavnije):
   Složenost hipoteza:
     standard: 1 (najjednostavnija)
     grue_2030: 2
     grue_2050: 2
     grue_2100: 2
   Odabrana: standard
   Razlog: Najniža složenost

3. Bayesov pristup (koristi priore):
   Prior vjerojatnosti:
     standard: 0.700
```

```
      grue_2030: 0.100
      grue_2050: 0.100
      grue_2100: 0.100
    Posterior (nakon opažanja):
      standard: 0.700
      grue_2030: 0.100
      grue_2050: 0.100
      grue_2100: 0.100
    Odabrana: standard
    Razlog: Najviši posterior

4. Ansambl metoda (kombinira hipoteze):
    Težinski prosjek predviđanja:
    Za 2025: 100.0% zeleno, 0.0% plavo
    Za 2040: 90.0% zeleno, 10.0% plavo
    Za 2060: 80.0% zeleno, 20.0% plavo
    Za 2110: 70.0% zeleno, 30.0% plavo
    Predviđanje: Ponderirana kombinacija

Zaključak: Praktična rješenja koriste dodatne
kriterije izvan čiste logike za izbor hipoteza.
```

9.1.9 Zaključak

Kroz ovu bilježnicu istražili smo duboku vezu između **Goodmanovog novog problema indukcije** i **No-Free-Lunch teorema** u strojnom učenju:

Ključni uvidi:

1. **Beskonačnost hipoteza**: Za svaki skup podataka postoji beskonačno mnogo jednako dobro podržanih hipoteza
2. **Nužnost biasa**: Bez induktivnog biasa, učenje je nemoguće - to nije bug već feature!
3. **NFL kao formalizacija**: No-Free-Lunch teoremi su matematička formalizacija Goodmanovog filozofskog argumenta
4. **Evolucija i bias**: Ljudska sposobnost učenja možda uspijeva zbog evolucijski oblikovanih biasa
5. **Praktična rješenja**: Regularizacija, Occamova oštrica i Bayesov pristup nude načine izbora među hipotezama

Filozofske implikacije:

Goodmanov problem pokazuje da **čista logika nije dovoljna** za induktivno zaključivanje. Trebamo dodatne kriterije - jednostavnost, priore, domensko znanje - koji nisu čisto logički.

Implikacije za AI:

Svaki sustav umjetne inteligencije mora riješiti Goodmanov problem implicitnim ili eksplicitnim izborom induktivnog biasa. **Nema univerzalnog algoritma učenja** - uspjeh ovisi o podudaranju između biasa algoritma i strukture problema.

Kao što Goodman zaključuje:

> "Valjanost induktivnog zaključka nije stvar logike već stvar povijesti uporabe predikata."

Možda je ključ uspješnog strojnog učenja u tome da naučimo kako odabrati prave biase za prave probleme - lekcija koju evolucija uči već milijunima godina.

Dio III

DODACI

Dodatak A

Uvod u Python za studente filozofije i ostalih ne-tehničkih grupa

A.1 Zašto bi se filozof zanimao za programiranje?

Na prvi pogled, svijet filozofije i svijet programiranja mogu se činiti kao dva potpuno odvojena svemira. Jedan se bavi vječnim pitanjima o smislu, postojanju i vrijednostima, dok se drugi bavi preciznim uputama za strojeve. Međutim, ispod površine, ova dva svijeta dijele duboke i iznenađujuće veze. Logika, temeljni alat filozofske analize, ujedno je i srce svakog računalnog programa. Način na koji strukturiramo argumente, definiramo pojmove i izvodimo zaključke u filozofiji ima svoj odraz u načinu na koji pišemo kod.

Učenje programskog jezika `Python`, stoga, za studenta filozofije nije samo stjecanje tehničke vještine, već i prilika za istraživanje poznatih koncepata iz nove perspektive. Kroz `Python`, apstraktni pojmovi poput varijabli, uvjeta i petlji postaju konkretni alati s kojima možete raditi, eksperimentirati i stvarati.

Ovo poglavlje je osmišljeno kao blagi uvod u `Python`, posebno prilagođen studentima humanističkih i društvenih znanosti. Nećemo se baviti složenim matematičkim problemima niti dubokim tehničkim detaljima. Umjesto toga, fokusirat ćemo se na osnove jezika, koristeći primjere koji su vam bliski: analizu teksta, rad s riječima i rečenicama, te istraživanje ideja kroz kod. Koristit ćemo se **Jupyter bilježnicama**, interaktivnim okruženjem koje omogućuje pisanje koda, teksta i vizualizacija na jednom mjestu, čineći učenje intuitivnim i zabavnim.

Dok budete prolazili kroz ovo poglavlje, potičem vas da ne gledate na kod samo kao na niz naredbi, već kao na novi način izražavanja i strukturiranja misli. Možda ćete otkriti da vam učenje programiranja može pomoći da postanete precizniji u svom filozofskom promišljanju, jasniji u svom izražavanju i kreativniji u svom pristupu problemima. Dobrodošli u svijet `Python`a!

A.2 Osnovni pojmovi: Varijable, tipovi podataka i izrazi

A.2.1 Varijable: Imenovanje ideja

U filozofiji, često koristimo simbole ili nazive kako bismo predstavili složene ideje. Na primjer, u logici, slovo P može predstavljati propoziciju "Svi ljudi su smrtni". Na sličan način, u Pythonu koristimo **varijable** kao imenovane spremnike za pohranu podataka.

Definicija A.1. Varijabla je imenovani prostor u memoriji koji služi za pohranu vrijednosti. Ime varijable (identifikator) koristimo kako bismo pristupili pohranjenoj vrijednosti.

Varijablu možete zamisliti kao oznaku koju pridružujete nekoj vrijednosti. Operator dodjele, znak jednakosti (=), koristi se za dodjeljivanje vrijednosti varijabli.

Dodjeljivanje vrijednosti varijablama.

```
pozdrav = "Zdravo, svijete!"
godinarodenjakanta = 1724
pipriblizno = 3.14159
```

U ovom primjeru, `pozdrav`, `godinarodenjakanta` i `pipriblizno` su nazivi varijabli. Jednom kada definiramo varijablu, možemo je koristiti u daljnjem kodu, na primjer za ispis njezine vrijednosti pomoću ugrađene funkcije `print()`.

```
print(pozdrav)
print(godinarodenjakanta)
```

Izlaz:

```
Zdravo, svijete!  1724
```

A.2.2 Tipovi podataka: Različite vrste informacija

U filozofiji, razlikujemo različite vrste pojmova: konkretne, apstraktne, pojedinačne, opće. Slično tome, u Pythonu, svaka vrijednost pripada određenom **tipu podataka**. Osnovni tipovi podataka koje ćemo za početak koristiti su:

- **String (`str`):** Niz znakova, odnosno tekstualni podaci. Stringovi se uvijek pišu unutar navodnika (jednostrukih `'` ili dvostrukih `""`). Primjeri: `"Sokrat"`, `'Platonova Država'`.
- **Integer (`int`):** Cijeli brojevi, bez decimalnog dijela. Primjeri: `42`, `-399`, `2025`.
- **Float (`float`):** Brojevi s pomičnim zarezom (decimalni brojevi). Primjeri: `3.14`, `9.81`, `-0.5`.
- **Boolean (`bool`):** Logička ili istinitosna vrijednost. Može imati samo dvije vrijednosti: `True` (istina) ili `False` (laž).

Python je dinamički tipiziran jezik, što znači da ne moramo unaprijed deklarirati tip varijable. Interpretator automatski prepoznaje tip podatka kada dodijelimo vrijednost. Tip varijable možemo provjeriti pomoću ugrađene funkcije `type()`.

Provjera tipova podataka.

```
filozof = "Aristotel"
    godinarodenja = -384
    visinaumetrima = 1.7
    jeliziv = False

    print(type(filozof))
    print(type(godinarodenja))
    print(type(visinaumetrima))
    print(type(jeliziv))
```

Izlaz:

```
<class 'str'> <class 'int'> <class 'float'> <class 'bool'>
```

A.2.3 Izrazi: Kombiniranje vrijednosti

U logici, kombiniramo propozicije pomoću veznika ($\wedge, \vee, \rightarrow$) kako bismo stvorili složenije izraze. U Pythonu, **izrazi** su kombinacije vrijednosti, varijabli i operatora koje se izračunavaju (evaluiraju) kako bi proizvele novu vrijednost.

- **Aritmetički izrazi:** Koriste standardne matematičke operatore (+, -, *, /).

```
a = 10
b = 5
zbroj = a + b
print(zbroj)
```

Izlaz:

```
15
```

- **String izrazi:** Operator + se može koristiti za spajanje (*konkatenaciju*) stringova.

```
ime = "Immanuel"
prezime = "Kant"
punoime = ime + " " + prezime
print(punoime)
```

Izlaz:

```
Immanuel Kant
```

A.3 Strukture podataka: Organiziranje misli

Dok osnovni tipovi podataka predstavljaju pojedinačne vrijednosti, **strukture podataka** služe za organiziranje i pohranu više vrijednosti u jednoj varijabli.

A.3.1 Liste: Uređeni nizovi argumenata

U filozofskim tekstovima, često nailazimo na nabrajanja ili nizove ideja, poput Aristotelovih četiriju uzroka. U Pythonu, **liste** su strukture podataka koje nam omogućuju pohranu uređenog niza elemenata. Elementi liste se navode unutar uglatih zagrada `[]`, odvojeni zarezima.

Definicija A.2. Lista (`list`) je promjenjiva, uređena kolekcija elemenata. "Uređena" znači da elementi zadržavaju redoslijed kojim su dodani. "Promjenjiva" znači da možemo dodavati, uklanjati ili mijenjati elemente nakon što je lista stvorena.

Kreiranje i pristupanje elementima liste.

```
aristoteloviuzroci = ["materijalni", "formalni", "djelatni", "svršni"] #
↪  Popiy Aristotelovih uzroka

# Pristupanje elementima pomoću indeksa
# Indeksiranje počinje od 0!
prviuzrok = aristoteloviuzroci[0]
treciuzrok = aristoteloviuzroci[2]

print("Prvi uzrok je:", prviuzrok)
print("Treći uzrok je:", treciuzrok)
```

Izlaz:

```
Prvi uzrok je:  materijalni Treći uzrok je:  djelatni
```

Liste su promjenjive. Možemo im dodavati nove elemente metodom `append()` ili uklanjati postojeće metodom `remove()`.

```
aristoteloviuzroci.append("imaginarni") # Dodavanje petog, "imaginarnog"
↪  uzroka
print(aristoteloviuzroci)

# Uklanjanje "imaginarnog" uzroka
aristoteloviuzroci.remove("imaginarni")
print(aristoteloviuzroci)
```

Izlaz:

```
['materijalni', 'formalni', 'djelatni', 'svršni', 'imaginarni'] ['materijalni',
'formalni', 'djelatni', 'svršni']
```

A.3.2 Rječnici: Asocijativni parovi pojmova i definicija

U filozofiji, često definiramo pojmove tako da im pridružujemo njihove definicije. U Pythonu, **rječnici** (`dict`) omogućuju pohranu podataka u obliku parova **ključ-vrijednost**. Ključevi su jedinstveni i koriste se za pristup pripadajućim vrijednostima.

Definicija A.3. Rječnik (`dict`) je promjenjiva, neuređena kolekcija parova ključ-vrijednost. Svaki ključ mora biti jedinstven unutar rječnika.

Rječnici se definiraju unutar vitičastih zagrada , a parovi ključ-vrijednost odvojeni su dvotočkom.

Kreiranje i korištenje rječnika.

```
filozofskirjecnik = {
    "epistemologija": "grana filozofije koja se bavi znanjem",
    "metafizika": "grana filozofije koja se bavi prvim uzrocima i principima
    ↪  bića",
    "etika": "grana filozofije koja se bavi moralom"
}

# Pristupanje vrijednosti pomoću ključa
definicijaetike = filozofskirjecnik["etika"]
print(definicijaetike)

# Dodavanje novog para
filozofskirjecnik["logika"] = "znanost o metodama i principima ispravnog
↪  zaključivanja"
print(filozofskirjecnik["logika"])
```

Izlaz:

```
grana filozofije koja se bavi moralom znanost o metodama i principima ispravnog
zaključivanja
```

A.4 Kontrola toka: Usmjeravanje argumentacije

Programi se ne izvršavaju uvijek linearno, od prve do zadnje naredbe. **Kontrola toka** odnosi se na naredbe koje nam omogućuju da usmjeravamo tijek izvršavanja programa, donosimo odluke i ponavljamo operacije.

A.4.1 Uvjetno izvršavanje: `if`, `elif`, `else`

U filozofskoj argumentaciji, često koristimo uvjetne rečenice oblika "Ako P, onda Q". U Pythonu, **uvjetne naredbe** nam omogućuju da izvršimo određeni dio koda samo ako je zadovoljen neki uvjet. Uvjet je izraz koji se evaluira kao `True` ili `False`.

Korištenje `if-else` strukture.

```
tvrdnja = "Sokrat je smrtan"

if "Sokrat" in tvrdnja:
    print("Tvrdnja se odnosi na Sokrata.")
else:
```

```
    print("Tvrdnja se ne odnosi na Sokrata.")
```

Izlaz:

```
Tvrdnja se odnosi na Sokrata.
```

Možemo koristiti i `elif` (skraćeno od *else if*) za provjeru više uzastopnih uvjeta.

```
godina = 1804

if godina < 476:
    print("Antička filozofija")
elif 476 <= godina < 1500:
    print("Srednjovjekovna filozofija")
else:
    print("Moderna i suvremena filozofija")
```

Izlaz:

```
Moderna i suvremena filozofija
```

A.4.2 Ponavljanje: `for` petlja

Često je potrebno ponoviti istu radnju više puta. Na primjer, analizirati svaku riječ u rečenici. U Pythonu, `for` **petlja** nam omogućuje da iteriramo (prolazimo) kroz elemente sekvence (poput liste ili stringa) i za svaki element izvršimo određeni blok koda.

Iteriranje kroz listu pomoću `for` petlje.

```
stoickevrline = ["mudrost", "pravednost", "hrabrost", "umjerenost"]

print("Prema stoicima, temeljne vrline su:")
for vrlina in stoickevrline:
    print("- " + vrlina)
```

Izlaz:

```
Prema stoicima, temeljne vrline su:  - mudrost - pravednost - hrabrost -
umjerenost
```

U ovom primjeru, varijabla `vrlina` se naziva *varijabla petlje*. U svakom prolasku (iteraciji) kroz petlju, ona poprima vrijednost sljedećeg elementa iz liste `stoickevrline`.

A.5 Funkcije: Modularizacija i ponovna upotreba misli

U filozofiji, kompleksne ideje često razlažemo na manje, razumljivije dijelove. U Pythonu, **funkcije** nam omogućuju da grupiramo niz naredbi u logičku cjelinu koju možemo pozvati više puta. Time se izbjegava ponavljanje koda i programi postaju organiziraniji i lakši za čitanje.

Definicija A.4. Funkcija je imenovani blok koda koji izvršava određeni zadatak. Može primiti ulazne podatke (argumente) i vratiti izlaznu vrijednost.

Funkcije definiramo pomoću ključne riječi `def`.

Definiranje i pozivanje jednostavne funkcije.

```
def pozdravifilozofa(ime):
    """
    Ova funkcija ispisuje pozdrav filozofu čije je ime
    proslijeđeno kao argument.
    """
    print("Pozdrav, " + ime + "!")

# Pozivanje funkcije
pozdravifilozofa("Platon")
pozdravifilozofa("Nietzsche")
```

Izlaz:

```
Pozdrav, Platon!  Pozdrav, Nietzsche!
```

Tekst unutar trostrukih navodnika odmah nakon definicije funkcije naziva se *docstring* i služi kao dokumentacija funkcije.

Funkcije mogu vraćati vrijednost pomoću naredbe `return`. Vraćena vrijednost se tada može pohraniti u varijablu ili koristiti u daljnjim izrazima.

Funkcija koja vraća vrijednost.

```
def sastaviime(ime, prezime):
    """Sastavlja puno ime iz dva dijela."""
    return ime + " " + prezime

punoimefilozofa = sastaviime("Simone", "de Beauvoir")
print(punoimefilozofa)
```

Izlaz:

```
Simone de Beauvoir
```

A.6 Primjer iz prakse: Analiza filozofskog teksta

Sada ćemo primijeniti sve što smo naučili na konkretnom primjeru: analizi kratkog filozofskog teksta. Cilj nam je prebrojati koliko se puta svaka riječ pojavljuje u poznatoj Descartesovoj izreci.

Brojanje riječi u tekstu.

```python
tekst = "Mislim, dakle jesam. Jesam, dakle postojim." # Korak 1: Definiramo tekst za analizu
print("Originalni tekst:", tekst)

# Korak 2: Priprema teksta
# Pretvaramo sva slova u mala slova kako 'Mislim' i 'mislim' ne bi bili različite riječi
tekstmali = tekst.lower()
# Uklanjamo interpunkcijske znakove
tekstbeztocke = tekstmali.replace('.', '')
tekstcisti = tekstbeztocke.replace(',', '')
print("Očišćeni tekst:", tekstcisti)

# Korak 3: Tokenizacija - razdvajanje teksta u listu riječi
rijeci = tekstcisti.split()
print("Lista riječi:", rijeci)

# Korak 4: Brojanje riječi pomoću rječnika
brojacrijeci = {}
for rijec in rijeci:
    if rijec in brojacrijeci:
        # Ako riječ već postoji u rječniku, povećaj brojač za 1
        brojacrijeci[rijec] = brojacrijeci[rijec] + 1
    else:
        # Ako je ovo prvo pojavljivanje riječi, dodaj je u rječnik s vrijednošću 1
        brojacrijeci[rijec] = 1

# Korak 5: Ispis rezultata
print("\nFrekvencija riječi:")
for rijec, broj in brojacrijeci.items():
    print(f"'{rijec}': {broj}")
```

Izlaz:

```
Originalni tekst: Mislim, dakle jesam. Jesam, dakle postojim. Očišćeni tekst:
mislim dakle jesam jesam dakle postojim Lista riječi: ['mislim', 'dakle',
'jesam', 'jesam', 'dakle', 'postojim']
Frekvencija riječi: 'mislim': 1 'dakle': 2 'jesam': 2 'postojim': 1
```

Ovaj primjer integrira varijable, stringove i njihove metode (`.lower()`, `.replace()`, `.split()`), liste, rječnike, `for` petlju i `if-else` uvjetnu logiku kako bi se riješio konkretan problem iz domene analize teksta.

Vježbe za poglavlje 1

Vježba A.1. Kreirajte rječnik koji sadrži pet vaših omiljenih filozofa kao ključeve, a njihove glavne filozofske ideje ili djela kao vrijednosti. Zatim, koristeći `for` petlju, ispišite svakog filozofa i njegovu ideju u formatu: `Ime Filozofa: Glavna ideja`.

Vježba A.2. Napišite funkciju pod nazivom `brojrijeci` koja prima jedan argument (string) i vraća broj riječi u tom stringu. (Savjet: metoda `split()` bi mogla biti korisna). Testirajte funkciju s nekoliko rečenica.

Vježba A.3. Napišite program koji provjerava pripada li godina određenom filozofskom raz-

doblju.

1. Definirajte listu koja sadrži nekoliko filozofa egzistencijalizma, npr. `egzistencijalisti = ["Sartre", "Camus", "Kierkegaard"]`.
2. Pitajte korisnika da unese ime filozofa pomoću funkcije `input()`.
3. Koristeći `if` naredbu i operator `in`, provjerite nalazi li se uneseno ime u vašoj listi.
4. Ispišite odgovarajuću poruku, npr. `Sartre je egzistencijalist.` ili `Platon nije egzistencijalist.`.

A.7 Zaključak: Sljedeći koraci

Ovo poglavlje pružilo je kratak pregled osnovnih elemenata programskog jezika Python. Vidjeli smo kako varijable i tipovi podataka predstavljaju osnovne gradivne blokove, kako strukture podataka poput lista i rječnika organiziraju informacije, kako kontrola toka usmjerava izvršavanje programa i kako funkcije omogućuju modularnost i ponovnu upotrebu koda.

Kao studenti filozofije, sada imate temelj za daljnje istraživanje. Sljedeći koraci mogli bi uključivati:

- **Rad s tekstom:** Python je izuzetno moćan za analizu teksta. Možete istražiti kako brojati riječi, analizirati sentiment, tražiti određene pojmove u velikim tekstualnim korpusima (npr. djelima pojedinih filozofa) i još mnogo toga. Biblioteke poput NLTK (Natural Language Toolkit) i spaCy otvaraju vrata svijeta *računalne lingvistike*.
- **Vizualizacija podataka:** Pomoću biblioteka kao što su Matplotlib i Seaborn, možete vizualizirati odnose između pojmova, učestalost riječi ili druge uvide koje dobijete analizom teksta, pretvarajući apstraktne podatke u jasne grafove.
- **Web scraping:** Možete naučiti kako automatski prikupljati tekstualne podatke s web stranica, na primjer, s filozofskih enciklopedija ili online arhiva.

Najvažnije je da se ne bojite eksperimentirati. Jupyter bilježnice su idealno okruženje za to. Pokušajte mijenjati primjere, postavljati si vlastite male probleme i tražiti rješenja. Programiranje, kao i filozofija, je vještina koja se razvija kroz praksu, znatiželju i upornost. Sretno s kodiranjem!

Dodatak B

Literatura

Pregled literature

Ova bibliografija obuhvaća temeljne radove iz područja logike, filozofije logike, računarske znanosti i njihovih međusobnih veza. Organizirana je u nekoliko tematskih cjelina koje prate strukturu udžbenika.

B.1 Klasični filozofski temelji logike

[Witt22] Wittgenstein, L. (1922). *Tractatus Logico-Philosophicus*. Trans. C. K. Ogden. London: Routledge. [Novija izdanja: Oxford 2023, ISBN: 978-0-19-886137-9; Penguin 2023, ISBN: 978-0-24-168195-4]

[Frege79] Frege, G. (1879). *Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens*. Halle: Louis Nebert.

[Frege84] Frege, G. (1884). *Die Grundlagen der Arithmetik*. Breslau: Wilhelm Koebner. [Engleski prijevod: *The Foundations of Arithmetic*, Northwestern University Press, ISBN: 0-8101-0605-1]

[Frege93] Frege, G. (1893/1903). *Grundgesetze der Arithmetik*. Jena: Hermann Pohle. [Engleski prijevod: *Basic Laws of Arithmetic*, Oxford 2013/2016, ISBN: 978-0-19-877730-4]

[RW10] Russell, B., & Whitehead, A. N. (1910-1913). *Principia Mathematica* (3 vols.). Cambridge: Cambridge University Press.

[Russ19] Russell, B. (1919). *Introduction to Mathematical Philosophy*. London: George Allen & Unwin. [Moderno izdanje: Routledge, ISBN: 0-415-09604-9]

[Tars56] Tarski, A. (1956). *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Trans. J. H. Woodger. Oxford: Clarendon Press. [Novo izdanje: Hackett 1983, ISBN: 0-915-14476-X]

B.2 Suvremeni udžbenici formalne logike

[**Mend15**] Mendelson, E. (2015). *Introduction to Mathematical Logic* (6th ed.). Boca Raton: CRC Press. ISBN: 978-1-48223-772-6.

[**Ende01**] Enderton, H. B. (2001). *A Mathematical Introduction to Logic* (2nd ed.). San Diego: Academic Press. ISBN: 0-12-238452-0.

[**vDal13**] van Dalen, D. (2013). *Logic and Structure* (5th ed.). London: Springer. ISBN: 978-1-44714-557-8.

[**Kune11**] Kunen, K. (2011). *Set Theory*. London: College Publications. ISBN: 978-1-84890-050-9.

[**Jech03**] Jech, T. (2003). *Set Theory* (3rd Millennium ed.). Berlin: Springer. ISBN: 3-540-44085-2.

B.3 Filozofija logike

[**Quin86**] Quine, W. V. (1986). *Philosophy of Logic* (2nd ed.). Cambridge, MA: Harvard University Press. ISBN: 978-0-674-66563-7.

[**Quin82**] Quine, W. V. (1982). *Methods of Logic* (4th ed.). Cambridge, MA: Harvard University Press. ISBN: 978-0-674-57176-1.

[**Haac78**] Haack, S. (1978). *Philosophy of Logics*. Cambridge: Cambridge University Press. ISBN: 0-521-29329-4.

[**Prie08**] Priest, G. (2008). *An Introduction to Non-Classical Logic: From If to Is* (2nd ed.). Cambridge: Cambridge University Press. ISBN: 978-0-521-67026-5.

[**Krip80**] Kripke, S. (1980). *Naming and Necessity*. Cambridge, MA: Harvard University Press. ISBN: 0-674-59845-8.

B.4 Teorija dokaza i prirodna dedukcija

[**Gent35**] Gentzen, G. (1935). Untersuchungen über das logische Schließen. *Mathematische Zeitschrift*, 39, 176-210, 405-431. [Engleski prijevod u: *The Collected Papers of Gerhard Gentzen*, North-Holland 1969]

[**Praw06**] Prawitz, D. (2006). *Natural Deduction: A Proof-Theoretical Study*. Mineola, NY: Dover. ISBN: 978-0486446554.

[**Take13**] Takeuti, G. (2013). *Proof Theory* (2nd ed.). Mineola, NY: Dover. ISBN: 978-0486490731.

[TS00] Troelstra, A. S., & Schwichtenberg, H. (2000). *Basic Proof Theory* (2nd ed.). Cambridge: Cambridge University Press. ISBN: 978-0521779111.

[Buss98] Buss, S. R. (Ed.). (1998). *Handbook of Proof Theory*. Amsterdam: Elsevier. ISBN: 978-0444898401.

[GLT89] Girard, J.-Y., Lafont, Y., & Taylor, P. (1989). *Proofs and Types*. Cambridge: Cambridge University Press. [Dostupno online]

B.5 Automatsko dokazivanje teorema

[CL73] Chang, C.-L., & Lee, R. C.-T. (1973). *Symbolic Logic and Mechanical Theorem Proving*. New York: Academic Press. [Dover reprint ISBN: 978-1493300242]

[Harr09] Harrison, J. (2009). *Handbook of Practical Logic and Automated Reasoning*. Cambridge: Cambridge University Press. ISBN: 978-0521899574.

[Fitt96] Fitting, M. (1996). *First-Order Logic and Automated Theorem Proving* (2nd ed.). New York: Springer. ISBN: 978-1461275152.

[NPW02] Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Berlin: Springer. LNCS 2283.

[BC04] Bertot, Y., & Castéran, P. (2004). *Interactive Theorem Proving and Program Development: Coq'Art*. Berlin: Springer.

B.6 Lambda račun i teorija tipova

[Bare85] Barendregt, H. P. (1985). *The Lambda Calculus: Its Syntax and Semantics* (Revised ed.). Amsterdam: North-Holland. ISBN: 978-0444875082.

[Pier02] Pierce, B. C. (2002). *Types and Programming Languages*. Cambridge, MA: MIT Press. ISBN: 978-0262162098.

[Pier04] Pierce, B. C. (Ed.). (2004). *Advanced Topics in Types and Programming Languages*. Cambridge, MA: MIT Press.

[Harp16] Harper, R. (2016). *Practical Foundations for Programming Languages* (2nd ed.). Cambridge: Cambridge University Press. ISBN: 978-1107150300.

[SU06] Sørensen, M. H., & Urzyczyn, P. (2006). *Lectures on the Curry-Howard Isomorphism*. Amsterdam: Elsevier. ISBN: 978-0444520777.

[ML84] Martin-Löf, P. (1984). *Intuitionistic Type Theory*. Naples: Bibliopolis. [Dostupno online]

[TD88] Troelstra, A. S., & van Dalen, D. (1988). *Constructivism in Mathematics* (2 vols.). Amsterdam: North-Holland.

B.7 Semantika programskih jezika

[**Wins93**] Winskel, G. (1993). *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA: MIT Press. ISBN: 978-0262731034.

[**Gunt92**] Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. Cambridge, MA: MIT Press. ISBN: 978-0262570954.

[**Stoy77**] Stoy, J. E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MA: MIT Press.

[**Reyn98**] Reynolds, J. C. (1998/2009). *Theories of Programming Languages*. Cambridge: Cambridge University Press. ISBN: 978-0521106979.

B.8 Logika u računarskoj znanosti

[**HR04**] Huth, M., & Ryan, M. (2004). *Logic in Computer Science: Modelling and Reasoning about Systems* (2nd ed.). Cambridge: Cambridge University Press. ISBN: 978-0521543101.

[**BA12**] Ben-Ari, M. (2012). *Mathematical Logic for Computer Science* (3rd ed.). London: Springer. ISBN: 978-1447141280.

[**NS93**] Nerode, A., & Shore, R. A. (1993). *Logic for Applications*. New York: Springer. ISBN: 978-0387948935.

[**Dijk76**] Dijkstra, E. W. (1976). *A Discipline of Programming*. Englewood Cliffs, NJ: Prentice-Hall.

B.9 Logičko programiranje

[**SS94**] Sterling, L., & Shapiro, E. (1994). *The Art of Prolog: Advanced Programming Techniques* (2nd ed.). Cambridge, MA: MIT Press. ISBN: 978-0262193382.

[**CM03**] Clocksin, W. F., & Mellish, C. S. (2003). *Programming in Prolog: Using the ISO Standard* (5th ed.). Berlin: Springer. ISBN: 978-3540006787.

[**Lloy87**] Lloyd, J. W. (1987). *Foundations of Logic Programming* (2nd ed.). Berlin: Springer. ISBN: 978-3642831911.

[**Brat12**] Bratko, I. (2012). *Prolog Programming for Artificial Intelligence* (4th ed.). Harlow: Pearson.

B.10 Bayesovska vjerojatnost i induktivna logika

[**Jay03**] Jaynes, E. T. (2003). *Probability Theory: The Logic of Science*. Cambridge: Cambridge University Press. ISBN: 978-0521592710.

[**Pearl88**] Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems*. San Francisco: Morgan Kaufmann. ISBN: 978-1558604797.

[**Good83**] Goodman, N. (1983). *Fact, Fiction, and Forecast* (4th ed.). Cambridge, MA: Harvard University Press. ISBN: 978-0674290716.

[**Hack75**] Hacking, I. (1975). *The Emergence of Probability*. Cambridge: Cambridge University Press. ISBN: 978-0521318037.

B.11 Dodatna literatura za Python programiranje

[**Lutz13**] Lutz, M. (2013). *Learning Python* (5th ed.). Sebastopol, CA: O'Reilly. ISBN: 978-1449355739.

[**McK17**] McKinney, W. (2017). *Python for Data Analysis* (2nd ed.). Sebastopol, CA: O'Reilly. ISBN: 978-1491957660.

[**VPG16**] VanderPlas, J. (2016). *Python Data Science Handbook*. Sebastopol, CA: O'Reilly. ISBN: 978-1491912058.

[**Ras13**] Raschka, S. (2015). *Python Machine Learning*. Birmingham: Packt Publishing. ISBN: 978-1783555130.

B.12 Neki radovi hrvatskih autora

[**Dov12a**] Dovedan Han, Z. (2012). *Formalni jezici i prevodioci 1: Regularni izrazi, gramatike, automati*. Zagreb: Element.

[**Dov12b**] Dovedan Han, Z. (2012). *Formalni jezici i prevodioci 2: Sintaksna analiza*. Zagreb: Element.

[**Dov12c**] Dovedan Han, Z. (2012). *Formalni jezici i prevodioci 3: Prevođenje i primjene*. Zagreb: Element.

[**Lauc19**] Lauc, D. at al (2019). *Pojmovnik elementarne logike*. Zagreb: FF Press. DOI: 10.17234/9789531758253.

[**Šik87**] Šikić, Z. (1987). *Systems of Rules and Systems of Sequents* (Doctoral dissertation). University of Zagreb.

[**Šik91**] Šikić, Z. (1991). A characterization theorem for consequence relations. *Mathematical Logic Quarterly*, 37(25-32), 385-390.

B.13 Online resursi i repozitoriji

[Git25] Lauc, D. (2025). *Logika u kodu: Jupyter bilježnice i primjeri*. GitHub repozitorij. Dostupno na: https://github.com/ffzg-logika/logika-u-kodu

[SEP] Stanford Encyclopedia of Philosophy. Dostupno na: https://plato.stanford.edu/

[IEP] Internet Encyclopedia of Philosophy. Dostupno na: https://iep.utm.edu/

[nLab] nLab - Mathematics, Physics, Philosophy. Dostupno na: https://ncatlab.org/

[MetaM] Metamath Proof Explorer. Dostupno na: http://us.metamath.org/

[Coq] The Coq Proof Assistant. Dostupno na: https://coq.inria.fr/

[Isa] Isabelle Proof Assistant. Dostupno na: https://isabelle.in.tum.de/

[Lean] Lean Theorem Prover. Dostupno na: https://leanprover.github.io/

Napomena: Ova bibliografija kontinuirano se ažurira novim izdanjima i relevantnim radovima. Za najnovije verzije i dodatne materijale, konzultirajte GitHub repozitorij udžbenika.