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.