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.