Univerzitet Crne Gore
Elektrotehnički fakultet

Prof. dr Vesna Popović-Bugarin

PyKnow – uparivanje šablona ograničavači vrijednosti polja

Vezni ograničavači vrijednosti polja

- **Primjer 1. Pronaći sve ljude koji nemaju braon kosu.**
- Jedan način da se ovo učini jeste da se koriste ograničavači polja kojima se ograničavaju vrijednosti koje varijabla može dobiti u LHS pravila.
- Jedna vrsta ograničavača polja su **tzv. vezni ograničavači**. Ovako se nazivaju jer služe za povezivanje varijabli i ograničenja vrijednosti koja neka polja mogu dobiti.
- Postoje tri vrste veznih ograničavača:
  1. Negacija (NOT) ~
  2. ILI (OR) |
  3. I (AND) &

Negacija – Primjer 1

- Negacija utiče na jedan uslov, vrijednost ili element koji slijedi odmah iza njega.
- Ukoliko vrijednost nekog ključa/elementa zadovoljava uslov koji slijedi iza negacije ili je jednaka vrijednosti koja je specificirana iza negacije, negacija nije zadovoljena za činjenicu kojoj taj ključ/element pripada. Ukoliko vrijednost ključa/elementa ne zadovoljava uslov naveden nakon negacije ili je različita od specificirane vrijednosti, negacija je zadovoljena.
- Negacija djeluje na vrijednost (ili uslov) koja slijedi, tako što ne dozvoljava da dati ključ ili element poprimi tu vrijednost (ili ispuni taj uslov).

```
from pyknow import *
class Osoba(Fact):
    pass
```

Primjer 1

```
class Upari(KnowledgeEngine):
    @DefFacts()
    def NekiLjudi(self):
        yield(Osoba(ime = "Marko Markovic", godine = 35, kosa = "plava",
                    oci = "plave"))
        yield(Osoba(ime ="Lazar Andric", godine = 27, oci = "crne",
                    kosa = "crna"))
        yield(Osoba (ime = "Mirko Markovic", godine = 18, oci = "zelene",
                     kosa = "braon"))
        yield(Osoba (ime = "Igor Ivanovic", godine = 28, oci = "crne",
                     kosa = "plava"))
    @Rule(AS.f1 << Osoba(ime = MATCH.ime,kosa = ~L("braon")))
    def NemaBraonKosu(self,ime,f1):
        print("%s nema braon kosu, ima %s kosu \n" %(ime,f1['kosa']))
```

```
engine = Upari()
engine.reset()
engine.run()
```

```
Igor Ivanovic nema braon kosu, ima plava kosu.
Lazar Andric nema braon kosu, ima crna kosu.
Marko Markovic nema braon kosu, ima plava kosu.
```

ILI ograničavač polja

- Koristi se da dozvoli da jedna ili više mogućih vrijednosti zadovolji neko polje šablona u uslovu. Dakle, ograničavač ili djeluje na dvije vrijednosti (ili dva uslova) koje povezuje, tako što dozvoljava da odgovarajuća vrijednost ključa/element činjenice poprimi bilo koju od tih vrijednosti (ili ispuni bilo koji od tih uslova).
- **Primjer 2. Naći sve ljude sa plavom ili braon kosom koristeći ili ograničavač polja.**

```
@Rule(AS.f1 << Osoba(ime = MATCH.ime,kosa = L("braon") | L("plava")))
def PlavaBraonKosa(self,ime,f1):
    print("%s ima %s kosu." %(ime,f1['kosa']))
```

Primjer 2

```
engine = Upari()
engine.reset()
engine.run()
```

```
Igor Ivanovic ima plava kosu.
Mirko Markovic ima braon kosu.
Marko Markovic ima plava kosu.
```

I – AND ograničavač polja

- Obično se koristi samo sa drugim ograničavačima, inače nema praktične koristi.
- Koristi se, u kombinaciji sa drugim ograničavačima, da se istovremeno na elegantan način promjenljiva veže za vrijednost koja zadovoljava ograničenje koje slijedi u nastavku. Dakle, ako je u prethodnom slučaju potrebno dalje utvrditi tačnu boju kose, rješenje je u vezivanju te boje za neku varijablu uz ograničenje da može primiti samo vrijednosti braon ili plava. Na ovaj način se izbjegava potreba za uzimanjem adrese činjenice.

```
@Rule(Osoba(ime = MATCH.ime,
            kosa = MATCH.boja1 & (L("braon") | L("plava"))))
def PlavaBraonKosa(self,ime,boja1):
    print("%s ima %s kosu." %(ime,boja1))
```

I – primjer

```
engine = Upari()
engine.reset()
engine.run()
```

```
Igor Ivanovic ima plava kosu.
Mirko Markovic ima braon kosu.
Marko Markovic ima plava kosu.
```

Primjer 3

- **Primjer 3. Naći osobe koje nemaju ni braon, ni plavu kosu i ispisati koje im je boje kosa.**

```
@Rule(Osoba(ime = MATCH.ime,
                    kosa = MATCH.boja &~L("braon") & ~L("plava")))
def NIBraonNiCrna(self,ime,boja):
    print("%s ima %s kosu." %(ime,boja))
```

```
engine = Upari()
engine.reset()
engine.run()
```

```
Lazar Andric ima crna kosu.
```

Kombinovanje ograničavača polja

- **Primjer 4. Provjeriti da li postoje dvije osobe takve da prva osoba ima ili plave ili zelene oči i nema crnu kosu. Druga osoba nema istu boju očiju kao prva osoba i ima ili crvenu kosu, ili joj je boja kose ista kao prvoj osobi.**

Primjer 4...

```
@Rule(Osoba(ime = MATCH.ime1,
            oci = MATCH.oci1 &(L("plave") | L("zelene")),
            kosa = MATCH.kosa1 & ~L("crna")),
      Osoba(ime = MATCH.ime2,
            oci = MATCH.oci2 & ~MATCH.oci1,
            kosa = MATCH.kosa2 & (L("crvena") | MATCH.kosa1)))
def DvijeOsobe(self,ime1,ime2,kosa1,kosa2,oci1,oci2):
    print("%s ima %s kosu i %s oci." %(ime1,kosa1,oci1))
    print("%s ima %s kosu i %s oci." %(ime2,kosa2,oci2))
```

```
engine = Upari()
engine.reset()
engine.run()
```

```
Marko Markovic ima plava kosu i plave oci.
Igor Ivanovic ima plava kosu i crne oci.
```

Ograničavači polja – komentari

- ***Promjenljiva mora biti već vezana ukoliko se koristi kao dio ili ograničavača.***

- ***Varijable se vezuju za neku vrijednost samo ako se pojavljuju samostalno ili su vezane sa drugim uslovima & ograničavačem polja.***

Funkcije i izrazi u PyKnow-u

- U PyKnow-u se mogu izvršavati matematičke operacije.
- PyKnow obezbijeđuje elementarne matematičke operatore +,-,*,/
- Uključivanjem biblioteke math - `from math import *` mogu se koristiti standardne funkcije
  - **sqrt** vraća kvadratni korijen argumenta.
  - **sum**, **fsum** daju sumu elemenata liste koja im je argument.
  - **max** vraća svoj maksimalni argument.
  - **min** vraća svoj minimalni argument.
  - **fabs** vraća apsolutnu vrijednost svog argumenta.
  - **fmod** vraća ostatak pri dijeljenju prvog argumenta drugim argumentom (koristi se pri radu sa realnim brojevima) % se koristi pri radu sa cijelim brojevima.
  - **float** konvertuje cio broj u realan i vraća njegovu vrijednost.
  - **integer** konvertuje realan broj u cio i vraća njegovu vrijednost

Primjer 4

- Napisati pravilo kojim se sumira površina pravougaonika. Visina i širina pravougaonika su ključevi činjenice Pravougaonik.

```
class Pravougaonik(Fact):
  pass
class suma(Fact):
  pass
```

```
class SumaPravougaonika(KnowledgeEngine):
  @DefFacts()
  def Pravougaonici(self):
    yield(Pravougaonik(sirina=3,visina=5))
    yield(Pravougaonik(sirina=2,visina=2))
    yield(Pravougaonik(sirina=1,visina=2))
    yield(Pravougaonik(sirina=3,visina=1))
    yield(suma(0))
  @Rule(Pravougaonik(sirina = MATCH.a,visina = MATCH.b),
          AS.f1 << suma(MATCH.suma1))
  def Sumiraj(self,a,b,suma1,f1):
    self.retract(f1)
    suma1 += a*b
    self.declare(suma(suma1))
    print("Nova suma je %s" %suma1)
```

Suma pravougaonika – pravilo

Ovo pravilo bi dovelo do ulaska u beskonačnu petlju.

```
engine = SumaPravougaonika()
engine.reset()
engine.run(5)
```

```
Nova suma je 3
Nova suma je 6
Nova suma je 9
Nova suma je 12
Nova suma je 15
```

Uvođenjem privremene činjenice koja pamti trenutnu površinu i razdvajanjem pravila na dva nova, izbjegava se beskonačna petlja.

```
class Pravougaonik(Fact):
  pass
class suma(Fact):
  pass
class Povrsina(Fact):
  pass
```

Sumiranje pravougaonika - revidirano

```
class SumaPravougaonika(KnowledgeEngine):
  @DefFacts()
  def Pravougaonici(self):
    yield(Pravougaonik(sirina=3,visina=5))
    yield(Pravougaonik(sirina=2,visina=2))
    yield(Pravougaonik(sirina=1,visina=2))
    yield(Pravougaonik(sirina=3,visina=1))
    yield(suma(0))
  @Rule(Pravougaonik(sirina = MATCH.a,visina = MATCH.b))
  def PovrsinaNova(self,a,b):
    self.declare(Povrsina(a*b))
  @Rule(AS.f1 << Povrsina(MATCH.P),AS.f2 << suma(MATCH.suma1))
  def Sumiraj(self,suma1,P,f1,f2):
    self.retract(f1)
    self.retract(f2)
    s = suma1 + P
    self.declare(suma(s))
    print("Nova suma je %s" %(s))
```

U RHS pravila se može uvoditi nova promjenljiva i vezati za vrijednost izraza.

Sumiranje pravougaonika - revidirano

```
engine = SumaPravougaonika()
engine.reset()
engine.run()
```

```
Nova suma je 3
Nova suma je 5
Nova suma je 9
Nova suma je 24
```

Složeni šabloni

- Do sada smo u pravilu navodili šablone koji su morali zadovoljavati navedena ograničenja. Moglo se javiti više šablona u istom pravilu, odvajali smo ih zarezom i podrazumijevali da svi moraju biti zadovoljeni prisustvom odgovarajućih činjenica.
- Za proizvoljno povezivanje šablona, koriste se uslovni elementi:
  - AND – kreira šablon koji je zadovoljen kada su zadovoljeni svi šabloni na koje se odnosi – isto kao zarez.
  - Primjer: Pravilo je zadovoljeno ukoliko su prisutne dvije činjenice, i to jedna koja zadovoljava šablon Fact(1) i druga koja zadovoljava šablon Fact(2).

```
@Rule(AND(Fact(1), Fact(2)))
 def _():
    pass
```

Složeni šabloni

- OR – kreira složeni šablon koji je zadovoljen kada je zadovoljen bilo koji od šablona na koje se odnosi
- Primjer: Pravilo je zadovoljeno ukoliko su prisutne bilo činjenica koja zadovoljava šablon Fact(1) i druga koja zadovoljava Fact(2)

```
@Rule(OR(Fact(1), Fact(2)))
def _():
    pass
```

- Ukoliko OR šablon zadovoljava više činjenica, pravilo u kojem se on nalazi će biti izvršeno više puta, i to po jedan put za sve činjenice koje ga zadovoljavaju.

Složeni šabloni

- NOT – Korišćenjem ovog uslovnog elementa kreira se šablon koji će biti zadovoljen ukoliko šablon na koji se odnosi ne zadovoljava nijedna činjenica, ili kombinacija činjenica
- Primjer: Šablon je zadovoljen ukoliko nijedna činjenica ne zadovoljava šablon Fact(1)

```
@Rule(NOT(Fact(1)))
def _():
    pass
```

I/O funkcije

Čitanje informacija sa tastature

- U PyKnow-u se mogu čitati informacije koje korisnik ukucava sa tastature korišćenjem `input()` funkcije.
- Osnovna sintaksa

`input('komentar')`

- Ukoliko bi se tražilo očitavanje imena sa tastature, pravilo bi bilo:

```
from pyknow import *
class Osoba(Fact):
  pass
class UnesiIme(KnowledgeEngine):
  @Rule(NOT(Osoba(W())))
  def PozdraviKorisnika(self):
    a = self.declare(Osoba(ime = input('Unesite ime i prezime')))
    print("Zdravo %s" %a["ime"])
```

input()

```
engine = UnesiIme()
engine.reset()
engine.run()
```

```
Unesite ime i prezimeVesna Popovic
Zdravo Vesna Popovic
```

Izvlačenje slamki – STICKS

- Igra za dva igrača.
- Na početku igre se smatra da postoji određen broj slamki na gomili. Svaki igrač, kada je njegov red, uzima jednu, dvije ili tri slamke.
- Cilj je izbjeći uzimanje zadnje slamke. Gubi onaj koji uzme posljednji slamku.
- Trik za pobjeđivanje u ovoj igri je u tome da se protivnik može prisiliti da izgubi ukoliko je vaš red i ostale su dvije, tri ili četiri slamke.
- Igrač koji ima pet slamki kada je na redu nema šansu da pobijedi, ukoliko njegov protivnik ne pogriješi.
- **Način da se protivniku ostavi pet slamki kada je on na redu, jeste da se uvijek ostavlja pet slamki, plus cjelobrojni umnožak broja četiri nakon vašeg reda. Dakle, na kraju vašeg reda, potrebno je da na gomili bude 5, 9, 13, 17, itd. slamki.**

Izvlačenje slamki – STICKS

- U našem slučaju program igra protiv čovjeka.
- Potrebno je odrediti ko će prvi da igra i kolika je početna veličina gomile slamki.
- Unošenje sa tastature i korišćenje dobijenih informacija u programu.
- Prisustvo činjenica `Stanje("biraj igraca")` će se kontrolisati aktiviranje tri vezana pravila:
- Prvo pravilo *BirajIgraca* kojim se od korisnika traži da odabere ko će da započne igru i koje će uvoditi privremenu činjenicu *Odabir_igraca* u koju će se smještati korisnikov unos i koja će sa činjenicom *Stanje* aktivirati drugo pravilo *PravilanOdabir* kojim se provjerava korisnikov unos i unosi činjenica *Na_potezu* sa jednim elementom koji pamti pravilan odabir i dalje će služiti za utvrđivanje ko je na redu (čovjek ili računar).

Izvlačenje slamki – STICKS

```
class Stanje(Fact):
  pass
class Odabir_igraca(Fact):
  pass
class Na_potezu(Fact):
  pass
class Gomila(Fact):
  pass
```

- Sada bi dio igrice do utvrđivanja prihvatanja odabira bio:

```
class Slamke(KnowledgeEngine):
  @DefFacts()
  def PocetnoStanje(self):
    yield(Stanje("Biraj igraca"))
    yield(Gomila(15))
  #Odabir Igraca na pocetku
  @Rule(Stanje("Biraj igraca"))
  def BirajIgraca(self):
    self.declare(Odabir_igraca(input("Ko igra prvi (Kompjuter: K Covjek: C): ")))
    #unos pomocne cinjenice sa jednim elementom koji odredjuje
    #igraca koji pocinje igru
```

Izvlačenje slamki – STICKS

- Nakon prhvatanja odabira, pravilo koje se aktivira ispravnim unosom je:

```
@Rule(AS.f1 << Stanje("Biraj igraca"),
      AS.f2 << Odabir_igraca(MATCH.igrac & (L("K") | L("C"))))
      # ukoliko je odabrano slovo C ili K
def PravilanOdabir(self,f1,f2,igrac):
    self.retract(f1)
    self.retract(f2)
    self.declare(Na_potezu(igrac))
    # cinjenicu Na_potezu cemo dalje koristiti za utvrdjivanje
    # igraca ciji je red
```

- Jedan ispravan unos bi bio:

```
engine = Slamke()
engine.reset()
engine.run()
```

```
Ko igra prvi (Kompjuter: K Covjek: C): K
```

Izvlačenje slamki – STICKS

- Nakon čega imamo činjenice sa veličinom gomile i informacijom o tome ko je na potezu i spremni smo da započnemo igru:

```
engine.facts
```

```
FactList([(0, InitialFact()), (2, Gomila(15)), (4, Na_potezu('K'))])
```

- Međutim, prije započinjanja igre, mora se definisati šta da se radi u slučaju lošeg odabira igrača. Naime, potrebno ponavljati unos sve dok se ne unese pravilna vrijednost. Potrebno je napraviti neku petlju kojom se pravilo za odabir igrača reaktivira sve dok se ne izvrši pravilan unos podataka. Dakle, pišemo pravilo koje reaktivira pravilo Biraj-igraca u slučaju kada je igrač pogrešno odabran. Za to je dovoljno da se izbaci i ubaci činjenica *Stanje("biraj-igraca")*.

Izvlačenje slamki – STICKS

```
@Rule(AS.f1 << Stanje("Biraj igraca"),
      AS.f2 << Odabir_igraca(MATCH.igrac & ~L("K")& ~L("C")))
def NepravilanOdabir(self,f1,f2):
  self.retract(f1)
  self.retract(f2)
  self.declare(Stanje("Biraj igraca"))
  #moramo ovu cinjenicu izbaciti i ponovo ubaciti u BZ jer stara
  #ne moze dva puta da aktivira isto pravilo
  print("Lose ste odabrali. Odaberite K ili C")
```

```
engine = Slamke()
engine.reset()
engine.run()
```

```
Ko igra prvi (Kompjuter: K Covjek: C): I
Lose ste odabrali. Odaberite K ili C
Ko igra prvi (Kompjuter: K Covjek: C): K
```

- Sada smo spremni za igru!
- Znamo ko je na potezu i sa koliko se slamki raspolaze

Izvlačenje slamki – STICKS

- Pretpostavimo da je korisnik dobro unio početnog igrača, ili da smo u toku igre i da je red na čovjeka.
- Ima smisla da čovjek bira koliko će uzeti slamki, samo ako na gomili ima više od jedne slamke.
- U tom slučaju čovjek se pita koliko slamki bira.
- Mora odabrati cio broj, i to 1, 2 ili 3, i odabran broj ne može biti veći od broja slamki koje se trenutno nalaze u gomili.
- Ovo cemo razbiti na tri pravila, i to prvo koje prihvata covjekov potez, i po jedno za pravilan i nepravilan covjekov potez
- Uvodimo i novu činjenicu

```
class Covjek_uzima(Fact):
  pass
```

Izvlačenje slamki – STICKS

```
@Rule(Na_potezu("C"),Gomila(MATCH.velicina&P(lambda x : x>1)))
def UzmiCovjekovPotez(self):
  self.declare(Covjek_uzima(int(input("Koliko slamki uzimate"))))
  #uvodimo cinjenicu koja cuva informaciju o broju slamki koje covjek
  #uzima, sluzice za provjeru pravilnosti njegovog unosa
  #Na_potezu("C") ne izbacujemo dok ne provjerimo unos
```

- Pretvaramo unos u cio broj i nećemo provjeravati taj dio

```
#Da bi odabir broja slamki bio ispravan, covjek mora odabrati cio broj,
#vec smo ga pri unosu pretvorili u cijeli broj.
#i to 1, 2 ili 3, i odabran broj ne može biti
#veći od broja slamki koje se trenutno nalaze u gomili.
@Rule(AS.f1 << Na_potezu("C"),Gomila(MATCH.velicina),
    AS.f2 << Covjek_uzima(MATCH.x),
    TEST(lambda x,velicina : ((x==1)|(x==2)|(x==3))&(x<=velicina)))
def PravilanCovjekovPotez(self,x,f1,f2):
  print("Pravilan covjekov potez. Uzeo je %s" %x)
  self.retract(f1)
  self.retract(f2)
```

Izvlačenje slamki – STICKS

```
@Rule(Na_potezu("C"),Gomila(MATCH.velicina),
    AS.f2 << Covjek_uzima(MATCH.x),
    TEST(lambda x,velicina : ((x<1)|(x>3))|(x>velicina)))
def NepravilanCovjekovPotez(self,x,f2):
  print("Nepravilan covjekov potez. Ponovite potez")
  self.retract(f2)
  self.declare(Covjek_uzima(int(input("koliko slamki uzimate"))))
```

```
engine = Slamke()
engine.reset()
engine.run()
```

```
Ko igra prvi (Kompjuter: K Covjek: C): C
Koliko slamki uzimate5
Nepravilan covjekov potez. Ponovite potez
koliko slamki uzimate2
Pravilan covjekov potez. Uzeo je 2
```

Izvlačenje slamki – STICKS

- U programu Slamka za određivanje broja slamki koje kompjuter treba da uzme sa gomile poslužićemo se činjenicom *Uzmi_slamku*, koja će imati dva ključa: *koliko* – kojim ćemo definisati koliko slamki kompjuter treba da se uzme i *ostatak* – ostatak prilikom dijeljenja trenutne veličine gomile sa četiri i koji će nam pomoći da kompjuter odluči koliko uzima.

```
class Uzmi_slamku(Fact):
  pass
```

- DefFact ćemo sada izmijeniti tako da čuva informacije koje koristi kompjuter pri odabiru svog poteza

```
    yield(Uzmi_slamku(koliko = 1,ostatak = 1))
    yield(Uzmi_slamku(koliko = 1,ostatak = 2))
    yield(Uzmi_slamku(koliko = 2,ostatak = 3))
    yield(Uzmi_slamku(koliko = 3,ostatak = 0))
  # zelimo da ostatak nakon naseg uzimanja bude
  # jedan, dakle ukupno cjelobrojan umnozak broja
  # cetiri u pet slamki
```

Izvlačenje slamki – STICKS

```
#jedini uslov koji mora biti ispunjen je da je velicina gomile veca od 1
@Rule(AS.f1 << Na_potezu("K"),Gomila(MATCH.velicina&P(lambda x : x >1)),
     Uzmi_slamku(koliko = MATCH.koliko,ostatak = MATCH.ostatak),
     TEST(lambda ostatak,velicina: ostatak == velicina % 4))
def KompjuterovPotez(self,koliko,f1):
  self.retract(f1)
  print("Kompjuter uzima %d" %koliko)
```