MÉRÉSI SEGÉDLET

Mesterséges Intelligencia Laboratórium

Egyszerű következtető rendszerek

Készítette:
Kovács Dániel László
Méréstechnika és Információs Rendszerek Tanszék

2003.

Tartalomjegyzék

I. SZAKÉRTŐI RENDSZEREKRŐL ÁLTALÁBAN **3**

II. SZABÁLY-ALAPÚ RENDSZEREK **5**

II.1. VISSZAFELÉ LÁNCOLT RENDSZEREK 5
II.2. ELŐREFELÉ LÁNCOLT RENDSZEREK 6

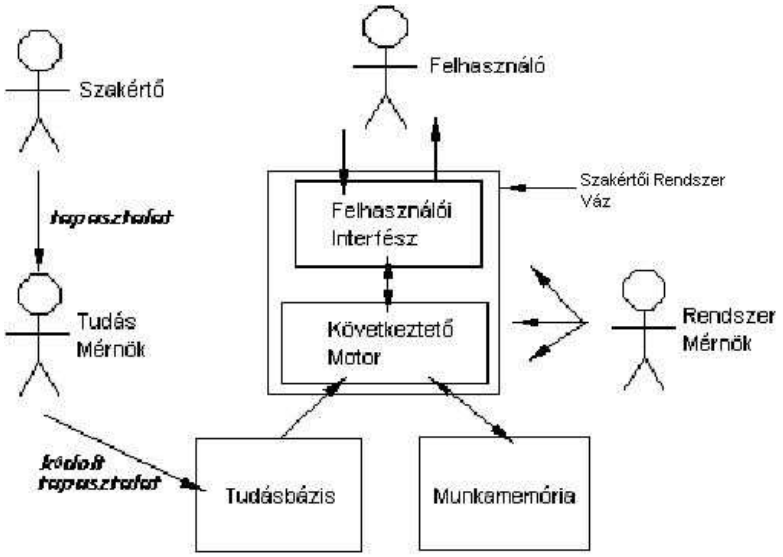
III. SZABÁLY-ALAPÚ SZAKÉRTŐI RENDSZEREK PROLOGBAN **7**

III.1. EGYSZERŰ VISSZAFELÉ LÁNCOLT SZAKÉRTŐI RENDSZER 7
III.2. BIZONYTALANSÁG-KEZELŐ VISSZAFELÉ LÁNCOLT SZAKÉRTŐI RENDSZER 22

I. Szakértői rendszerekről általában

A szakértői rendszerek olyan alkalmazások, melyek bizonyos típusú feladatok megoldására használható nem-algoritmikus tapasztalatot testesítenek meg. Sakkozhatnak, pénzügyi tervezést végezhetnek, számítógépeket konfigurálhatnak, valós idejű rendszereket monitorozhatnak, biztosítók stratégiáját támogathatják, vagy tetszőleges más olyan szolgáltatást végezhetnek, melyhez előzőleg emberi tapasztalat volt szükséges.

A szakértői rendszerek általános felépítése az II. 1. ábrán látható:



II. 1. Ábra: Szakértői rendszerek és emberi interfészeik

Értelmezzük a fenti ábrán vázolt rendszer-komponenseket!

- *Tudásbázis*: a tapasztalat egyfajta deklaratív reprezentációja, többnyire „HA-AKKOR" típusú szabályok formájában.
- *Munkamemória*: adott feladathoz tartozó, probléma-specifikus adathalmaz.
- *Következtető motor*: a rendszer magjában elhelyezkedő azon funkcionális elem, mely következtetést végez a tudásbázis és a munkamemória probléma-specifikus adatai alapján.
- *Felhasználói interfész:* a felhasználó és a rendszer közt zajló dialógus vezérléséért felelős komponens.

Hasonlóan kategorizálhatók a rendszerhez kapcsolódó egyéni szerepkörök is:

- *Szakértő*: azon szakértő(k), aki(k) azoknak a feladatoknak a megoldásáért felelős(ek), melyek megoldására a szakértői rendszert tervezzük.
- *Tudásmérnök:* a szakértő(k) tapasztalatát a szakértői rendszer által felhasználható, deklaratív formába átültető egyén.

- *Felhasználó:* azon egyén, aki olyan tanácsért fordul a rendszerhez, amiért máskülönben a szakértőt kellene felkeresnie.

A felhasználói interfészt és a következtető motort együttes formában szokás a szakértői rendszer *vázának* is nevezni. A tudásmérnök tehát egy-egy vázat használ arra, hogy adott problématerületre alkalmazható szakértői rendszert építsen. Természetesen előfordulhat az is, hogy nem csak a tudásbázist, hanem már magát a vázat is egy-egy konkrét alkalmazáshoz fejlesztik. Ekkor egy újabb szerepkör is megjelenik a rendszerben:

- *Rendszermérnök:* azon egyén, aki felépíti a felhasználói interfészt, kidolgozza a tudásbázis deklaratív formátumát és implementálja a következtető motort.

A következőkben a szakértői rendszerek azon speciális változatát vizsgáljuk, melyekben a tudást tények és szabályok halmaza reprezentálja.

II. Szabály-alapú rendszerek

A szabály-alapú rendszerek[1] a tudást nem statikus formában, atomi kijelentések gyűjteményeként reprezentálják, hanem olyan szabályok halmazaként, melyek bizonyos összefüggésekről tesznek megállapítást. Ilyen megállapítás lehet pl. az, hogy adott helyzetben mi a teendő, vagy, hogy mi következik az adott szituációból, stb. A szabály-alapú rendszerek tehát HA-AKKOR típusú *szabályok* és *tények* halmazából, továbbá egy *interpreterből* tevődnek össze, amely a szabályok alapján, a tények függvényében alakítja a rendszer működését.

A szabály-alapú rendszereknek két főbb fajtáját különböztetjük meg: *előrefelé láncolt* és *visszafelé láncolt* rendszereket. Egy <u>előrefelé láncolt rendszerben</u> tények egy halmazából indulunk ki, és a szabályok sorozatos alkalmazásával (tüzelésével) újabb tényekhez jutunk (vagy cselekvéseket hajtunk végre). Az újabb tények alapján – lehetőség szerint – tovább folytatjuk a működést. Egy <u>visszafelé láncolt rendszerben</u> egy hipotézisből (célból) indulunk ki, melyet a rendelkezésünkre álló tények alapján szeretnénk bizonyítani. Ehhez először is olyan szabály(oka)t keresünk, mely(ek)ből következtethetünk az adott hipotézisre. Amennyiben létezik ilyen szabály, és a feltétel része csupa ismert tény-állítás, akkor a hipotézis bizonyosságot nyert. Ha nem, akkor a talált szabály feltétel részét újabb (rész)hipotézis(ek)nek, avagy (rész)cél(ok)nak tekintve – lehetőség szerint – tovább folytatjuk a működést. Az előrefelé láncolt rendszereket elsősorban <u>adat-vezéreltnek</u>, míg a visszafelé láncolt rendszereket <u>cél-vezéreltnek</u> nevezzük. Lássuk először az utóbbiakat!

II.1. VISSZAFELÉ LÁNCOLT RENDSZEREK

Cél-vezérelt esetben, avagy visszafelé láncolt produkciós rendszerek esetén a problémákat célszerű „strukturált választási” problémaként modellezni, ahol a cél az, hogy a rendszer adott számú lehetőség közül a legmegfelelőbbet válassza.[2]

A tudást esetünkben tehát választásokat leíró szabályok reprezentálják. Ezek a problémát rész-problémákra bontják. Lássunk egy példát! Adottak a következő szabályok a tudásbázisban:

HA család = albatrosz
    ÉS szín = világos
        AKKOR madár = ’fehér albatrosz’

HA család = albatrosz
    ÉS szín = sötét
        AKKOR madár = ’fekete albatrosz’

[1] A szabály-alapú rendszerekre szokás produkciós rendszerekként is hivatkozni.
[2] Ily módon modellezhetők pl. a diagnosztikai rendszerek, melyek célja, hogy adott számú diagnózis közül a legmegfelelőbb válasszák ki.

E szabályok és a tudásbázisban aktuálisan szereplő – pl. látott madárra vonatkozó – tények alapján megpróbálhatjuk bebizonyítani, hogy a fekete albatroszról van-e szó. A rendszer tehát végigpróbálja mindazon szabályokat, melyek közvetlenül elősegítik a „madár = ’fekete albatrosz’” tény megállapítását. Esetünkben csak a második szabály jöhet szóba. Sajnos azonban (a példa kedvéért tegyük fel, hogy) tudásbázisunkban csupán a „szín = fekete” tény szerepel. A madár családjának meghatározása ennél összetettebb feladat, ezért egy újabb szabály felelős érte:

HA rend = csőorrú
    ÉS méret = nagy
        ÉS szárny = ’hosszú-vékony’
            AKKOR család = albatrosz

A problémát tehát rész-problémákra bontottuk, mivel ahhoz, hogy bebizonyítsuk, hogy fekete albatroszról van szó, azt kell bizonyítanunk, hogy „rend = csőorrú”, „méret = nagy”, „szárny = ’hosszú-vékony’” és természetesen „szín = fekete” igaz…

II.2. ELŐREFELÉ LÁNCOLT RENDSZEREK

Sajnos sok esetben nem oldható meg a lehetséges válaszok előzetes felsorolása. Így hát a rendszernek nem a válaszok felől „visszafelé” haladva kell megoldania a problémát, hanem a meglévő tények alapján kell tovább, és tovább „finomítania” a már meglévő megoldást. Adat-vezérelt esetben tehát nyomon kell követnünk a megoldás állapotát és további következtetést kell végeznünk a megfelelő szabályok kiválasztása és „tüzelése” által. Vegyük példának a következő – lakberendezésért felelős – előrefelé láncolt szabály-alapú rendszer tudásbázisát! Adottak a következő tények a munkamemóriában:

’TV nincs a helyén’
’Dívány az X falnál’
’Az X fal az Y fallal szemben helyezkedik el’

Illetve a következő szabály:

HA ’TV nincs a helyén’
    ÉS ’Dívány az X falnál’
        ÉS ’Az X fal az Y fallal szemben helyezkedik el’
            AKKOR Mozgat(TV, ’Y fal’)

A munkamemóriabeli tények alapján a szabály elsüthető, azaz végrehajtható a következmény része. Esetünkben ez most nem egy újabb tény munkamemóriába iktatását jelenti, hanem egy cselekvést, amely a TV-t a dívánnyal szemben, az Y falhoz helyezi el. Mivel a szituáció, azaz a munkamemóriabeli tények megváltoznak a szabály elsütésének hatására, ezért a szabály nem süthető el újra – a rendszer előrefelé láncolt következtetése leáll.

III. Szabály-alapú szakértői rendszerek Prologban

Ebben a fejezetben mind előre, mind visszafelé láncolt szabály-alapú szakértői rendszerekkel foglalkozunk. Megvalósításukhoz a Prolog nyelvet, azon belül a Sicstus Prolog-ot használjuk. Feladatunk mindkét esetben két lépésre lesz bontható: (1) szakértői (többnyire informális) tapasztalati tudás tudásbázissá alakítása, majd (2) következtető motor és esetleg felhasználói interfész, avagy szakértői rendszer-keret építése. Megvalósításaink megalkotásakor az előrefelé láncolt esetben a kifogástalan, hatékony működés megvalósítására helyezzük a hangsúlyt, míg visszafelé láncolt esetben a rendszer kezelhetőségére és sokoldalúságára. Kezdjük az utóbbival!

III.1. EGYSZERŰ VISSZAFELÉ LÁNCOLT SZAKÉRTŐI RENDSZER

Kezdetben adott egy informális szakértői tapasztalati tudás. Az egyszerűség kedvéért vegyük az MI tankönyv[3] jól ismert példáját (némileg módosítva):

„…bűnöző azon amerikai polgár, aki fegyvert ad el egy ellenséges országnak. A Nono ország ellenséges, azaz ellensége Amerikának, mint országnak. Ha Nono rakétákat birtokol, akkor ezeket a rakétákat mind csak az amerikai Kurtz ezredes adhatta el nekik. A rakéták fegyverek – tudhattuk meg az amerikai Dr.Strangelove ezredestől. Újabban az is kiderült, hogy Nono M1-es rakétákat birtokol…”

Az első lépés a szakértői rendszer létrehozására, amikor az informális tapasztalatot a tudásmérnök átülteti egy deklaratív – esetünkben szabály-alapú, Prolog-os – tudásbázisba. Próbáljuk meg a tudást elsőrendű logikai formában reprezentálni! Ekkor a tényállítások *Predikátum(Argumentum)* alakúak lesznek, s ilyen – ponált, vagy negált – tényállítások konjunkciója és/vagy diszjunkciója alkotja majd az implikációk, azaz következtetések feltétel és következmény részét, azaz a szabályokat. Milyen tények és szabályok nyerhetők ki a fenti informális leírásból?

Vezessük be a főbb fogalmakat úgy, hogy a predikátumok egy-egy attribútumot, míg a predikátumok argumentuma az adott attribútumok értékét reprezentálja: `attributum(ertek)`. Válasszuk ezt az egyszerű adatreprezentációs formát! A következő (számunkra fontosabb) fogalmak adottak a szövegben:

„X bűnöző”	→	`bunozo(X)`
„X amerikai polgár”	→	`amerikai(X)`
„X fegyver”	→	`fegyver(X)`
„X rakéta”	→	`raketa(X)`
„X ország”	→	`orszag(X)`
„X ellenséges”	→	`ellenseges(X)`

[3] Russell, S.J. és Norvig, P „Mesterséges Intelligencia modern megközelítésben”, PANEM, 2000.

A szövegben kapcsolatok is megjelennek. Ezeket célszerű relációkként reprezentálni:

„X ellensége Y-nak”	→	`ellensege(X,Y)`
„X birtokol Y-t”	→	`birtokol(X,Y)`
„X elad Y-t Z-nek”	→	`elad(X,Y,Z)`

Tehát eddig a szövegben előforduló attribútumokat, értéküket, továbbá relációkat fogalmaztunk meg általános formában. Ezeket a tényeket haszontalan lenne máris a munkamemóriába helyezni, hiszen argumentumaik mind változók, amelyek tetszőleges kifejezéssel egyesíthetők, s így akár tautológiának, egyúttal „értelmetlennek” is tekinthetnénk őket. Mondjuk inkább csak azt, hogy egy „modellt” alkottunk a szövegben foglalt tudás egy részének reprezentációjára. A tudás többi része ugyanis implikációk (és konkrét, behelyettesített tények) formájában reprezentálható. A HA-AKKOR alakú implikációk Prolog-ban ugyebár (AKKOR :- HA) alakú, fejjel és törzzsel rendelkező klózként ábrázolhatók. Vegyük tehát sorra a tudásbázisba iktatandó szabályokat!

„…bűnöző azon amerikai polgár, aki fegyvert ad el egy Amerikával ellenséges országnak.”

```
bunozo(X) :-
     amerikai(X),
     fegyver(Y),
     orszag(Z),
     ellenseges(Z),
     elad(X,Y,Z).
```

„A … ország ellenséges, azaz ellensége Amerikának…”

```
ellenseges(X) :-
     ellensege(X,amerika).
```

„Ha Nono rakétákat birtokol, akkor ezeket a rakétákat mind csak az amerikai Kurtz ezredes adhatta el nekik.”

```
elad(kurtz,X,nono) :-
     birtokol(nono,X),
     raketa(X).
```

„A rakéták fegyverek.”

```
fegyver(X) :-
     raketa(X).
```

Miután feltöltöttük a tudásbázist szabályokkal, helyezzük el a munkamemóriában az – ugyancsak szövegben említett – konkrét szituációra vonatkozó tényeket:

„A Nono ország…”

```
orszag(nono).
```

„…Amerikának, mint országnak.”

```
orszag(amerika).
```

„…Nono … ellensége Amerikának…”

```
ellensege(nono,amerika).
```

„…az amerikai Kurtz…”

```
amerikai(kurtz).
```

„…az amerikai Dr.Strangelove…”

```
amerikai(dr_strangelove).
```

„…M1-es rakétákat…”

```
raketa(m1).
```

„… Nono M1-es rakétákat birtokol…”

```
birtokol(nono,m1).
```

Az elkészült tudásbázis és a feltöltött, kiindulási állapotot tükröző munkamemória mellett már felvázolható az őket felhasználó szakértői rendszer terve. Szerencsére, első feladatunk esetében, azaz a visszafelé láncolt következtető motor létrehozása esetén nyugodtan használható a Prolog integrált következtetési mechanizmusa. Ez is épp visszafelé láncolt következtetést valósít meg, hiszen, ha meggondoljuk, a bizonyítani kívánt állításokat először a szóbajövő klózok fejével (most szabályok következmény részével) illeszti, majd sikeres illeszkedés esetén hasonlóan cselekszik a klóz törzsében (most szabályok feltétel részében) lévő állításokkal, mint esetleges újabb klóz-fejekkel. Legyen első bizonyítani kívánt hipotézisünk, mellyel szemléltetni szeretnénk „újdonsült” visszafelé láncolt következtető motorunk működését az, hogy Dr.Strangelove bűnöző!

Ehhez adjuk a Prolog interpreter promptjára a `bunozo(dr_strangelove)` kijelentést! Az eredmény a következő:

```
?- bunozo(dr_strangelove).
no
```

A Prolog jól láthatóan a `no` választ adta – magyarán megcáfolta hipotézisünk, miszerint jelenlegi tudásunkból következik, hogy Dr.Strangelove bűnöző… Node akkor ki bűnöző? Van egyáltalán? …nem marad tehát más, mint vagy találgatni, vagy megkérdezni magát a rendszert. A visszafelé láncolt következtetés szerencsére lehetőséget biztosít erre, hiszen a `bunozo(X)` kérdés hatására a Prolog gyakorlatilag a *„Ki a bűnöző?”* kérdésre adja meg a választ. A kérdés hatására az X változóba helyettesítve (pontosabban azzal *egyesítve*) áll elő a válasz, hiszen nincs olyan megkötés, hogy hipotézisünk teljesen konkrét legyen, avagy logikai reprezentációja teljesen *behelyettesített*. Az eredmény ekkor a következő:

```
?- bunozo(X).
X = kurtz? ;
no
```

A „;” jel beírásával további válaszokat próbáltunk volna kicsikarni kérdésünkre – hátha van még Kurtz ezredesen kívül más is, aki bűnöző… Jelenlegi tudásunk alapján azonban a visszafelé következtető motor, azaz a Prolog következtető mechanizmusa nem volt képes más plauzibilis behelyettesítést találni X változó értékére (pontosabban a *visszalépések* során az újabb behelyettesítések mind meghiúsultak). Így hát kiderült, hogy csak és egyedül Kurtz ezredes tekinthető bűnözőnek jelenlegi tudásunk (pontosabban tudásbázisunk és aktuális munkamemóriánk) alapján.

Mint láttuk, tudásunk nem csak tudásbázisunkban helyezkedik el, hanem a munkamemóriában is, tények formájában. Ha azt szeretnénk, hogy tudásunk egészében a tudásbázisban foglaljon helyet, akkor két főbb eset lehetséges: (1) áthelyezzük a jelenleg munkamemóriában elhelyezkedő tényeket a tudásbázisba, vagy (2) a tudásbázisban jelenleg elhelyezkedő szabályokat minden lehetséges módon behelyettesítjük úgy, hogy a munkamemória tényei alapján igaznak bizonyuljanak, majd az eddigi szabályokat a behelyettesített (változók nélküli) szabályokra cseréljük, a munkamemóriát pedig – tetszőleges mértékben – átmásoljuk a tudásbázisba, majd kiürítjük. A két változatból az utóbbi konkrétabb (ám persze kevésbe rugalmas) tudást tükröz, de legalább a kiindulási tudás nem oszlik meg a tudásbázis és a munkamemória közt[4]. Ígyhát a (2)-es átalakítást választjuk. Első lépésben csak az `ellensege(nono,amerika)` és a `birtokol(nono,m1)` tényeket helyezzünk át a munkamemóriából a tudásbázisba.. A transzformáció végrehajtása után a munkamemória tehát üres, a tudásbázis pedig a következő:

[4] A visszafelé láncolt rendszereknek általában nem szükséges munkamemóriabeli tudás, hiszen a tudásbázis alapján próbálunk valamit bebizonyítani. Ellenben az előrefelé láncolt rendszereknél szükséges egyfajta kezdeti munkamemóriabeli tudás megléte is ahhoz, hogy a következtetés működésbe léphessen.

```
bunozo(kurtz) :-
     amerikai(kurtz),
     fegyver(m1),
     orszag(nono),
     ellenseges(nono),
     elad(kurtz,m1,nono).

ellenseges(nono) :-
     ellensege(nono,amerika).

elad(kurtz,m1,nono) :-
     birtokol(nono,m1),
     raketa(m1).

fegyver(m1) :-
     raketa(m1).

ellensege(nono,amerika).

birtokol(nono,m1).
```

Innen tehát már „ránézésre" is látható, hogy mi a *„Ki a bűnöző?"* feladat megoldása. Ez tehát a szövegben foglalt informális tudás legkonkrétabb reprezentációja. A jövőben tehát, ha tudásmérnökünk szakértői tapasztalatot alakít tudásbázissá, remélhetőleg nyomban erre a **„lehetőleg behelyettesített szabályok"**-ból álló formára hozza… Sajnos azonban – bármily meglepő is – se a `bunozo(kurtz)` se a `bunozo(X)`, se semmilyen más hipotézis nem bizonyítható e tudás alapján – a tudásbázis nem használható következtetésre! Ennek nyilvánvaló oka, hogy hiányos (vagy nem megfelelő reprezentációjú) a rendszerbe foglalt tudás. Minek a reprezentációja rossz: mi hiányos? A válasz a (2)-es transzformáció végrehajtásában keresendő…

A létrejött hiányt a felhasználóval való kommunikáció oldhatja fel. A lényeg az, hogy amit a rendszer momentán nem tud (ami tudást a munkamemória törlésével elvesztettünk), mostantól meg kell majd kérdezni a felhasználótól! Meg kell kérdezni, hogy valóban igazak-e a nem összetett, törzs nélküli tényállítások amiket – a példa kedvéért – úgymond „elfelejtettünk"! A szakértői rendszer tehát, bár – a szabályok felépítéséből és behelyettesítéséből következően – tudja, hogy MIKOR-MIT kell kérdezni, az aktuális állapotról, észlelésről, avagy szituációról nincs teljes ismerete. Nem tudja, hogy Kurtz tényleg amerikai-e, vagy, hogy az M1 az egy rakéta-e, stb. Ezekre tehát jelen esetben mind-mind rá kell majd kérdeznie[5].

[5] Természetesen bizonyos munkamemóriabeli tények – (2)-es transzformációnak megfelelő – tudásbázisba másolásával megtartható egy ún. rész-tudás. Ekkor a rendszernek már nem kell minden apró-cseprő tényállítás igazságára rákérdeznie, elég csak esetleg a legfontosabbakra, amik – valami oknál fogva – még nem részei a rendszer tényszerű tudásának. Példánkban is feltételezzük, hogy a rendszer rendelkezik némi – állapotra vonatkozó – rész-tudással. Persze mondhatnánk, hogy a tudásmérnök nem végzett tökéletes munkát. Az is lehet, hogy az informális szöveg nem tartalmazta az összes információt. A lényeg, hogy a rendszer adott esetben a felhasználóra, a felhasználó szituációra vonatkozó ismereteire kell hagyatkozzon.

A „rákérdezés”, azaz a kommunikáció megvalósításához egy felhasználói interfészt kell definiálnunk a felhasználó és a következtető motor „közé” (lásd. II.1. ábra). A felhasználói interfész és a következtető motor (avagy együttesen *keret*) megtervezése a rendszermérnök dolga, így hát most az Ő szerepét vesszük magunkra. Kezdjük a következő, igen egyszerű változattal! Gyűjtsük össze a tudásbázis szabályainak feltétel részeiben szereplő azon, törzzsel nem rendelkező, egyszerű állításokat, melyek momentán még nem szerepelnek a tudásbázisban külön tényként, azaz, amelyekről nem tudjuk, hogy igazak-e! Ezek a következők:

```
orszag(amerika).

amerikai(kurtz).

raketa(m1).
```

Általánosítsuk az állításokat, azaz írjunk változót az argumentumukba, majd adjunk mindegyikhez egy törzset is a következőképp:

```
orszag(X) :-
     ask(orszag,X).

amerikai(X) :-
     ask(amerikai,X).

raketa(X) :-
     ask(raketa,X).
```

Az elkészült összetett tényeket helyezzük el a tudásbázisban! Ha tehát a rendszer valamely szabály visszafelé láncolt feldolgozásánál olyan egyszerű (azaz törzzsel nem rendelkező) állítás bizonyításába ütközött, amely nem szerepelt a tudásbázisban, most helyette egy olyan összetett (azaz törzzsel rendelkező), általános tényállításba fut bele, aminek törzse egy „kérdés”. Úgy is felfoghatjuk a dolgot, mintha az eddigi egyszerű, tudásbázisban nem szereplő tényállításokat szabályokká alakítottuk volna, melyek következmény része a bizonyítandó tény, feltétel része pedig egy *kérdés*. A tényt akkor tekintjük igaznak, ha a kérdés igaz. A kérdés igazságát a felhasználó dönti el – ez a válasz a kérdésre. Hozzuk tehát létre a megfelelő `ask/2` klózt[6]:

```
ask(X,Y) :-
     write(X:Y),
     write('? '),
     read(yes).
```

[6] A `pred/N` jelölés a `pred` nevű, N argumentumú predikátumra hivatkozik.

Most tehát a Prolog a `bunozo(X)` kérdésre pl. a következő interakciót folytathatja le a felhasználóval:

```
?- bunozo(X).
amerikai:kurtz? yes.
raketa:m1? yes.
orszag:nono? yes.
raketa:m1? yes.
X = kurtz ? ;
no
```

Példánkban a rendszer négy kérdést tett fel a felhasználónak, melyekre a felhasználó minden esetben igen-nel (Prologban `yes`) válaszolt. A rendszer a válaszokból arra következtetett, hogy Kurtz tábornok a bűnöző. További bűnözők a rendszer állítása szerint nem léteznek.

Furcsának tűnhet, hogy a rendszer kétszer is megkérdezte, hogy M1-es e a rakéta… Ennek oka, hogy mind az `elad/3`, mind a `fegyver/1` szabályok feltétel részében szerepel a `raketa/1`, sőt, konkrétan a `raketa(m1)` állítás. Módosítsuk tehát az `ask/2` klózt úgy, hogy képes legyen a felhasználó válaszainak feljegyzésére! A módosított változat lehet pl. a következő:

```
ask(X, Y):-
     known(yes, X, Y),
     !.

ask(X, Y):-
     known(_, X, Y),
     !,
     fail.

ask(X, Y):-
     write(X:Y),
     write('? '),
     read(Answer),
     assert(known(Answer, X, Y)),
     Answer == yes.
```

A tudásbázisban tehát a fenti 3 klózra kell lecserélnünk az eddigi 1-et. Lássuk részletesen a klózok működését! Az `ask/2`, ahogy eddig is, úgy most is, mindig csak behelyettesített X és Y argumentumokkal hívódik meg újdonsült szabályaink (mint például `raketa/1`) feldolgozásakor. Az első `ask/2` klóz azt vizsgálja meg, hogy szerepel-e a munkamemóriában a megfelelő X és Y argumentumokkal a `known/3` klóz. A `known/3` klóz reprezentálja tehát egy-egy „X attribútum Y érték” párra a felhasználó

eddig adott válaszát. Ha ez „igen” volt, azaz, ha a munkamemóriában `yes` első argumentummal szerepel a megfelelő X és Y párhoz tartozó known/3 klóz, akkor az azt jelenti, hogy a felhasználó már egyszer „igennel” válaszolt arra a kérdésre, hogy: *„Igaz-e, hogy X attribútum Y értékű?”*. Ha ez valóban így van, azaz a felhasználó – valamikor régebben már – valóban „igennel” felelt a megfelelő (X és Y)-hoz tartozó kérdésre, akkor egy *vágót* (jele: „!”) kell végrehajtani, hiszen a többi klóz ekkor már nem érdekes számunkra… A vágó hatására tehát a további (programkódban lentebb szereplő) `ask/2` klózokkal már nem foglalkozik a rendszer[7] - irrelevánsak. Tehát adott X és Y értékek mellett a `known(yes, X, Y)` egyfajta „őr” szerepét játszotta, amely eldönti, hogy végrehajtódhat-e a vágó, s így megszüntethető-e az `ask/2` klóz „meghívásakor” létrehozott „3-ágú” választási pont.

Az `ask/2` második klóza nagyon hasonló az elsőhöz. Meghívására csak akkor kerül sor, ha az első (fentebb taglalt) klóz meghiúsul, azaz, ha a megfelelő X és Y behelyettesítés mellett a `known(yes, X, Y)` klóz meghiúsul, avagy nem szerepel a munkamemóriában. Ekkor tehát elképzelhető, hogy, ha nem is `yes` első argumentummal, de valamilyen más tetszőleges első argumentummal (jele: „_”) azért mégiscsak szerepel a munkamemóriában. Ennek vizsgálatára szolgál a második `ask/2` klóz. Ha szerepel, akkor az nemleges választ jelent, azaz a felhasználó az adott X és Y párhoz tartozó megfelelő kérdésre „nemmel” válaszolt. Ekkor tehát a többi `ask/2` klózzal már nem is kell foglalkoznunk, lefuthat egy vágó, majd, mivel a felhasználó válasza „nem” volt a kérdésre, meg kell hiúsulnia az `ask/2` kérdésnek! Erre a Prologban a `true` kulcsszóval szemben (amely ugyebár a vágóhoz hasonlóan, ám *mellékhatás* nélkül, mindig teljesül), a `fail` kulcsszó használatos (amely mindig meghiúsul).

A harmadik, utolsó `ask/2` klózra csak akkor térünk rá, ha az előző kettő meghiúsul. Ez csak akkor lehetséges, ha a megfelelő X és Y behelyettesítéshez nem tartozik semmilyen `known/3` klóz sem a munkamemóriában, azaz, ha a felhasználó még nem válaszolt az *„Igaz-e, hogy X attribútum Y értékű?”* kérdésre. A harmadik `ask/2` klóz feladata tehát az, hogy a kérdést feltegye, a választ a munkamemóriában eltárolja, majd a válasznak megfelelően sikerüljön, vagy meghiúsuljon. Először tehát a szokott módon feltesszük a kérdést. A felhasználó válaszát az `Answer` változóval egyesítjük. Ezt követően az `assert` parancs segítségével beszúrjuk a megfelelő X és Y behelyettesítéshez tartozó, `Answer` választ eltárolni hivatott `known/3` állítást a munkamemóriába. Majd végül, ha az `Answer` változó értéke `yes`, akkor sikerül az `ask/2`, ha valami más, akkor nem…

Sajnos a fenti kód az első meghívás alkalmával „exception”-t jelez, s így nem hajlandó lefutni. Ennek oka, hogy a `known/3` predikátum dinamikus, s így kezdetben még nem

[7] A vágó mindig teljesül. Esetünkben jól láthatóan *vörös vágó*–ról van szó, hiszen illeszthető klózokat „vág le”, azaz olyan választási pontot szüntet meg, amely adott esetben valóban releváns választásokat (s így behelyettesítéseket, klóz-redukciókat) reprezentálhat. Tudjuk ugyebár, hogy a *zöld vágó* csak „életképtelen” választási pontokat szüntet meg, azaz gyakorlatilag előre jelzi a Prolog interpreter számára, hogy a többi azonos predikátum-nevű és argumentum-számú klózzal valóban nem érdemes foglalkozni, mivel azok majd úgyis mindig meghiúsulnának. …esetünkben tehát vörös vágókat helyeztünk el az `ask/2` klózókban…

szerepel a munkamemóriában, csak később, ha a felhasználó kérdéseinek nyomán beszúrjuk egy-két példányát. A Prolog tehát megkülönbözteti azokat a predikátumokat, amelyek nem léteznek azoktól, amelyeknek csupán csak nincs egy klóza sem. Jelezzük tehát a Prolog számára, hogy itt egy létező, ám kezdetben még klózok nélküli predikátumról van szó! Ehhez a programkód elején (a legeslegelején ☺) a

```
:- dynamic known/3.
```

kijelentést kell elhelyeznünk[8]. Tehát a `bunozo(X)` most már gond nélkül lefut, és a megfelelő választ adja eredményül. Ha a Prolog indítása, majd a tudásbázis és felhasználói interfész betöltése után tesszük fel a kérdést, akkor, mivel a munkamemória még teljesen üres, ezért az összes fontosabb kérdést felteszi számunkra. A munkamemóriában való válaszok eltárolása azonban nem csak a közvetlenül megismételt kérdésfeltevés elkerülésére alkalmas, hanem a közvetettre is. Ha újra feltesszük a `bunozo(X)` kérdést, akkor a rendszer már a teljes – általunk az előző kérdésfeltevés során megadott – tudás ismeretében nem kényszerül egy kérdés feltevésére sem: nyomban megadja a választ.

Most tegyük fel, hogy újabb ismeret érkezik a tudásbázisba. Az ismeret informális megfogalmazása a következő: *„Előfordulhat, hogy a rakéták eladásáért valójában Dr.Strangelove a felelős. Dr.Strangelove a rakéták eladásakor Kurtz ezredesnek adja ki magát. A csalást úgy lehet felismerni, hogy míg Kurtz ezredes szemszíne kék, addig Dr.Strangelove szemszíne fekete."* Vezessük tehát be a `szemszin(X)` fogalmat, ahol X a szemszín attribútum értéke (pl. kék, fekete, stb). A fogalom több változást is indukál a tudásbázisban. Egyrészt a `bunozo/1` klóz felépítése változik, hiszen szerepel benne a `szemszin/1` is. Másrészt a szemszín alapján különbséget tehetünk abban, hogy ki a bűnöző, hiszen immár Dr.Strangelove is felmerült potenciálisan. Tehát immár két `bunozo/1` klóz is lesz: az egyik Kurtz-ra vonatkozik, a másik Dr.Strangelove-ra. Tehát az eddigi egy `bunozo/1` klóz helyett a következő két `bunozo/1` klóz szerepel a tudásbázisban:

```
bunozo(kurtz) :-
     amerikai(kurtz),
     fegyver(m1),
     orszag(nono),
     ellenseges(nono),
     szemszin(kek),
     elad(kurtz,m1,nono).
```

[8] A „:-„-vel kezdődő, úgymond fejjel nem rendelkező állítások a programkód interpretálása, fordítása során végrehajtódnak, mintha csak szabályos Prolog-hívások lennének, amiket mi, mint felhasználók gépeltünk be a felhasználói felületen…

```
bunozo(dr_strangelove) :-
     amerikai(dr_strangelove),
     fegyver(m1),
     orszag(nono),
     ellenseges(nono),
     szemszin(fekete),
     elad(kurtz,m1,nono).
```

Ezen felül magát a `szemszin/1` klózt is be kell illeszteni a tudásbázisba:

```
szemszin(X) :-
     ask(szemszin,X).
```

Az `ask/2` klóz, avagy a felhasználói interfész változtatására jelenleg nincs szükség. Üres munkamemória mellett tehát tegyük fel újra a `bunozo(X)` kérdést a rendszernek, és lehetőség szerint mindenre igen-nel válaszoljunk. Ekkor a futás eredménye a következő:

```
?- bunozo(X).
amerikai:kurtz? yes.
raketa:m1? yes.
orszag:nono? yes.
szemszin:kek? yes.
X = kurtz ? ;
amerikai:dr_strangelove? yes.
szemszin:fekete? no.
no
```

A párbeszéd során a rendszer immár a „megfigyelt” szemszínre vonatkozó kérdést is feltett a felhasználónak. A párbeszéd során a felhasználó „kék”-nek vélte a rakétákat eladó amerikai szemszínét… A rendszer tehát első válaszként Kurtz ezredest jelölte meg. Ám a felhasználó ezt nem tekintette elegendőnek, így további válaszokat kért. A rendszer következő kérdése (*„Amerikai-e Dr.Strangelove?”*) már a második `bunozo/1` klóz következménye. A következő – bunozo/1 klózban szereplő – három állítás, a `fegyver/1`, az `orszag/1` és az `ellenseges/1` már az előző `bunozo/1` klóz illesztése során bizonyosságot nyert, s így a felhasználó válaszainak nyomán a megfelelő `known/1` klózok assert-álódtak a munkamemóriába. Ebből következően (lásd. `ask/2` felhasználói interfész) a rendszer már nem kérdez rá újra az ismert attribútum-érték párokra. Ámde a `szemszin(fekete)` attribútum-érték páros igazsága még nem ismert, ígyhát a rendszer felteszi erre vonatkozó kérdését. A felhasználó nem-mel válaszolt a „*Fekete-e a szemszín?”* kérdésre. Válasza érezhetően logikus, ígyhát elvárható

volna, hogy a Prolog magától kikövetkeztesse. Logikus azért, mert tudjuk (vagy tudhatnánk), hogy csak egyvalaki: vagy Kurtz, vagy Dr.Strangelove, vagy valaki más adhatta el a rakétákat. Az se biztos, hogy amerikai. Sőt, még az se biztos, hogy Kurtz, vagy Dr.Strangelove amerikai (ezért is kérdezi meg a rendszer). Annyi azonban biztos, hogy csak *egyvalaki* adta el a rakétákat. Ennek az egyvalakinek pedig csak egyféle szemszíne lehet: vagy kék, vagy fekete, vagy valami más! Tehát, ha a rendszer már egyszer tudja, hogy mi a szemszín, akkor felesleges újra megkérdeznie. Egyvalakinek ugyebár nem lehet egyszerre két szemszíne[9].

A fentiekben a lényeg a következő: vannak olyan attribútumok (pl. amerikai, ország, rakéta, stb), amiknek több értéke is lehet (pl. Kurtz és Dr.Strangelove is lehet amerikai, vagy Amerika és Nono is ország, stb). Ám vannak olyan attribútumok is (pl. szemszín), melyeknek – tudásmérnökünk szerint – csak egy értéke lehet (pl. vagy kék, vagy fekete, stb). A kétfajta attribútum-fajta megkülönböztetésére vezessük be a `multivalued/1` predikátumot a tudásbázisba. Ez a predikátum jelölje azokat az attribútumokat, melyeknek akár több ismert értéke is lehet. Ha valami nem többértékű, akkor biztosan egyértékű. A tudásbázisba tehát a következő állításokat kell beillesztenünk:

```
multivalued(orszag).
multivalued(amerikai).
multivalued(raketa).
```

Ezen felül ki kell egészítenünk a felhasználói interfészt, avagy az `ask/2` klózokat is egy újabb példánnyal, amely támogatja az egy-többértékűség megkülönböztetését! Ezen klóz feladata az, hogy az előző klózokon felül először vizsgálja meg, hogy egyértékű-e a vizsgált X attribútum, amire a kérdés irányul. Ha nem egyértékű, akkor a felhaszálói interfész eddigi funkcionalitása érvényesüljön, maradjon minden a régiben. Ha azonban egyértékű, akkor meg kell nézni, hogy létezik-e már ismert értéke az attribútumnak. Ha nem létezik, akkor minden folyik tovább úgy, ahogy eddig. Ha azonban létezik ismert értéke, amelyről tudjuk, hogy igaz, akkor meg kell vizsgálni, hogy a kérdés az ismert értékre irányul-e. Ha igen, akkor a folyamat megint csak nem változik, marad minden a régiben, avagy az eddigi első `ask/2` klóz kezeli le az esetet. Ha azonban a kérdés egy másik értékre irányulna, akkor tudva, hogy az attribútum egyértékű, a kérdésnek meg kell hiúsulnia! Tehát az eddig `ask/2` klózok elé a következő új első `ask/2` klózt kell beírni:

```
ask(X, Y):-
     \+ multivalued(X),
     known(yes, X, Y2),
     Y \== Y2,
     !,
     fail.
```

[9] Eltekintve a David Bowie-hoz hasonló esetektől…

A kiegészített tudásbázis, és a módosított felhasználói interfész mellett, üres munkamemóriával a `bunozo(X)` kérdésre az előbbi párbeszéd a következőképp alakul:

```
?- bunozo(X).
amerikai:kurtz? yes.
raketa:m1? yes.
orszag:nono? yes.
szemszin:kek? yes.
X = kurtz ? ;
amerikai:dr_strangelove? yes.
no
```

Látható, hogy a rendszer immár nem teszi fel feleslegesen az újabb, szemszínre vonatkozó kérdést. Ezt tehát az egy- és többértékű attribútumok megkülönböztetésének köszönhetjük. Megjegyezném, hogy a fenti párbeszédet még hatékonyabbá tehetnénk úgy, ha a `bunozo/1` klózokban a `szemszin/1` predikátumot nem az utolsó előtti helyen, hanem a legelső helyen szerepeltetnénk a konjunkciókban[10].

Mostani tudásbázisunknak van egy nagy hátránya: csak behelyettesített argumentumú predikátumok szerepelhetnek benne. Például jól látható, hogy az, hogy valaki bűnöző-e, nem csak az M1-es rakéta eladásának lehet a következménye, hanem tetszőleges rakéta eladásának. Jelenlegi tudásbázisunk nem képes kezelni ezt az esetet, azaz, ha kiderülne, hogy Nono-nak épp nem M1-es, hanem pl. MAVERICK rakétái vannak, akkor az illetőt, aki eladta őket – legyen az akár Kurtz, akár Dr.Strangelove – a rendszer nem tekintené bűnözőnek, hiszen nincs a szabályok közt! Módosítsuk tehát a `fegyver/1`, `bunozo/1` és `elad/3` klózokat a következőképpen:

```
fegyver(X) :-
     raketa(X).

bunozo(kurtz) :-
     amerikai(kurtz),
     fegyver(X),
     orszag(nono),
     ellenseges(nono),
     szemszin(kek),
     elad(kurtz,X,nono).
```

[10] Ennek oka a szekvenciális feldolgozás. Hiszen tudjuk, hogy amíg az „X = 1, var(X)” konjunkció meghiúsul, addig a „var(X), X = 1” konjunkció sikerrel lefut Prologban annak ellenére, hogy logikailag elvben nem lenne szabad semmilyen különbségnek sem lenni kettejük közt. A szekvenciális feldolgozás tehát egyaránt lehet előnyős, vagy éppen hátrányos…

```
bunozo(dr_strangelove) :-
     amerikai(dr_strangelove),
     fegyver(X),
     orszag(nono),
     ellenseges(nono),
     szemszin(fekete),
     elad(kurtz,X,nono).

elad(kurtz,X,nono) :-
     birtokol(nono,X),
     raketa(X).
```

Ámde most a probléma az, hogy a `raketa/1` attribútummal kapcsolatos kérdés a rendszer részéről meghatározatlan értékkel (azaz változóval, behelyettesítetlenül) érkezik! Nincs értelme azt kérdezni, hogy *„Rakéta-e X?”* úgy, hogy *„X”* bármi lehet, akár rakéta is… Ezért ebben az esetben a rendszernek célszerű volna felajánlania a lehetséges válaszadási lehetőségeket, *„X”* lehetséges értékeit, mint válaszokat. Ehhez szükséges az eddigi „igen/nem” alapú kérdések feltevésére alkalmas felhasználói interfész további kiegészítése, csiszolása. Ezentúl tehát nem csak „igen/nem” alapú kérdéseket kell tudnunk feltenni, hanem menüben is kell tudnunk ajánlani a felhasználó lehetséges válaszadási lehetőségeit. Szükség van tehát egy kiegészített `ask/2` klózra is! Nevezzük ezt `menuask/3`-nak:

```
menuask(X,Y,_) :-
     known(yes,X,Y),
     !.

menuask(X,Y,MenuList) :-
     write('What is the value for '),
     write(X),
     write('?'),
     nl,
     write(MenuList),
     nl,
     read(A),
     member(A, MenuList),
     assert(known(yes,X,A)),
     Y = A.
```

A `menuask/3` működése lényegében nagyon hasonló az `ask/2` működéséhez. Ha szerepel az attribútum-érték pár a munkamemóriában, akkor sikerül. Ha változó értékre kérdezünk rá, akkor abban az esetben is sikerül, ha legalább egy értékét ismerjük az adott argumentumnak. Ha az első klóz nem teljesül, akkor felteszi a menü szerinti kérdést, és megvizsgálja, hogy a válasz szerepel-e a menüben. Ha igen, akkor (mint releváns

attribútum-értéket) beszúrja a munkamemóriába (azaz megjegyzi, mint igaz választ), majd megvizsgálja, hogy a konkrét esetben egyesíthető-e a kérdezett érték a felhasználó által adottal. Ha nem, akkor meghiúsul, ha igen, akkor a felhasználó vagy megfelelő értéket adott, vagy a kérdezett érték nem volt konkrét (avagy változó volt). A második klóz érdekessége, hogy a LISTS könyvtár `member/2` predikátumát használja működéséhez. Az említett könyvtárat úgy tölthetjük be, ha a forráskód elejére a következő sort írjuk:

```
:- use_module(library(lists)).
```

Végezetül ahhoz, hogy mindez működésbe lépjen, a `raketa/1` klózban az `ask/2` hívást `menuask/3`-ra kell cserélni:

```
raketa(X) :-
     menuask(raketa,X,[m1,maverick,sidewinder]).
```

Adott tehát a tudásbázis (`bunozo/1, ellenseges/1, elad/3, fegyver/1, ellensege/2, birtokol/2, szemszin/1, amerikai/1, orszag/1, raketa/1, multivalued/1`), a felhasználói interfész (`ask/2, menuask/3`), a munkamemória (`known/3`) és a következtető motor (Prolog sajátja). Ahhoz, hogy a fenti komponenseket teljes értékű szakértői rendszernek lehessen tekinteni, keretbe kell foglalnunk a felhasználói interfészt a következtető motorral! Hozzunk létre egy általános `top_goal/1` predikátumot a tudásbázisban, amely a szakértői rendszer által aktuálisan vizsgált legfőbb kérdést fogalmazza meg:

```
top_goal(X) :-
     bunozo(X).
```

A tudásbázis betöltését tételezzük fel automatikusnak. Ezt egyébként a következtető motorban illene megvalósítani. Így hát most a következtető motor feladata kizárólag a feladat – tudás alapján történő – megoldása. Erre a solve/0 eljárás szolgálhat:

```
solve :-
     abolish(known),
     assert(known(1,2,3)),
     retract(known(1,2,3)),
     top_goal(X),
     write('The answer is '),
     write(X),
     nl.
```

```
solve :-
     write('No answer found.'),
     nl.
```

A `solve` üríti ki a munkamemóriát, hívja meg a `top_goal/1`-et, majd írja ki az eredményt. Most pedig kapcsoljuk össze a következtető motort a felhasználói felülettel, pl. a következő felhasználói felületbe integrált „rutin”-ok által:

```
go :-
     greeting,
     repeat,
     write('> '),
     read(X),
     do(X),
     X == quit.

greeting :-
     write('Visszafele   lancolt   szabaly-alapu   szakertoi
     rendszer.'),
     nl,
     write('Parancsok: consult, quit.'), nl.

do(consult) :-
     solve,
     !.

do(quit).

do(X) :-
     write(X),
     write(' nem megfelelo parancs.'), nl,
     fail.
```

Tehát a szakértői rendszer indításához a **„go”** utasítást kell begépelni a szükséges forrásfájlok betöltése után. A `go/0` eljárás a `do/1` eljárást használja, amely gyakorlatilag a szakértői rendszer parancskészletének magas szintű feldolgozásáért felelős „high-level interpreter”. Ezzel tehát elkészült egy tetszőleges tudásbázishoz és munkamemóriához illeszthető, egyszerűbb, visszafelé láncolt következtetést megvalósító, szabály-alapú szakértői rendszer!

III.2. BIZONYTALANSÁG-KEZELŐ VISSZAFELÉ LÁNCOLT SZAKÉRTŐI RENDSZER

Ebben a fejezetben egy olyan visszafelé láncolt következtetést megvalósító következtető motort építünk, mely képes a bizonytalanság kezelésére is. Ehhez elsősorban a következtetési szabályok formátumának megváltoztatására, azaz a tudás-reprezentáció módosítására lesz szükség. Foglalkozunk bizonytalan következményű, és a felhasználói bizonytalanság nyomán bizonytalan feltétel részű szabályokkal. Egy Prolog implementáción mutatjuk be a működés elveit.

Először is definiáljuk a szabályok formátumát! Legyen a szabályoknak egy egyedi azonosító neve (`Name`), egy bal (`LHS`), és egy jobb oldala (`RHS`). Ekkor a szabályokat a következő `rule/3` predikátumok reprezentálhatják a tudásbázisban:

```
rule(Name,LHS,RHS).
```

A szabályok – visszafelé láncolt következtetés során – bizonyítandó jobb oldalait tehát a `rule/3` predikátum harmadik argumentuma tartalmazza. Legyen ennek formája a következő:

```
rhs(Goal, CF)
```

Tehát a `rule/3` harmadik argumentuma egy `rhs/2` predikátum. Ennek első argumentuma egy újabb predikátum, nevezetesen a `rule/3` által reprezentált szabály jobb oldalán elhelyezkedő attribútum-érték[11] páros. Ezen párosok általános reprezentációjára vezessük be az `av/2` predikátumot a következőképp:

```
av(Argument, Value)
```

[11] Az *attribútum-érték* alakú adatreprezentációnál sokkal összetettebb, kifejezőbb adatreprezentációs formák is léteznek. Egy szinttel bonyolultabbnak tekinthető az *objektum-attribútum-érték* hármas. Itt egy-egy konkrét objektum egy-egy konkrét argumentumának egy-egy értékét egy-egy két-argumentumú predikátum reprezentál. Ha egy-egy objektumból (pl. székből, asztalból, stb) több is lehet, illetve az említett objektum példányoknak több attribútuma is lehetséges, akkor a *rekordok* használata javasolt. Itt tehát egy-egy objektum-példányt egy-egy N+1 argumentumú predikátum reprezentál, melynek első argumentuma a konkrét objektum-példány megnevezése (azonosítója), míg a maradék N argumentum az adott objektum attribútumait reprezentálja. Rekordok egy (körmentes) irányított gráfját *kereteknek (frames)* tekinthetjük. Az objektum orientált struktúrákhoz hasonlóan, itt is adottak szülők és gyermekek, melyek öröklik a szülők tulajdonságait (leszármazás). Egy rekord gyermekei azon rekordok, melyekbe irányított él vezet az adott rekordból. Ez a rekord tekinthető az említett rekordok szülőjének. A gyermek rekordok öröklik a szülő rekord N argumentumát, továbbá kiegészítik sajátjukkal is, stb. A kereteknek további érdekes tulajdonsági is vannak, amik vizsgálata kívül esik a segédlet tárgykörén – külön labor keretében foglalkozunk velük. Összefoglalva: Adatreprezentációs felosztásunkban a keretek tekinthetők a legösszetettebb, legkifejezőbb adatreprezentációs formának.

Az `av/2` predikátumok mindkét argumentuma atomi. Tehát amit eddig `a(b)` reprezentált a III.1. szakaszban, azt most `av(a,b)` reprezentálja. Az `rhs/2` predikátum második (`CF`)[12] argumentuma a `rule/3` által reprezentált szabály következmény-részének bizonytalanságát hivatott számszerűsíteni. A `rule/3` predikátumok második (`LHS`) argumentuma az adott predikátum által reprezentált szabály bal oldalát, azaz feltétel részét reprezentálja. Ennek alakja legyen a következő:

```
lhs(GoalList)
```

Az `lhs/1` predikátum egyetlen (`GoalList`) argumentuma egy `av/2` predikátumokból álló lista, amely tehát a `rule/3` szabályok feltétel részében szereplő attribútum-érték párok konjunkcióját, avagy a szabály feltétel részét hivatott reprezentálni. A `rule/3` szabály neve (`Name`) egyszerűen csak egy atom[13]. Lássuk tehát az előbbi fejezet (teljesen behelyettesített) szabályait és azok bizonytalanságot is magába foglaló változatait! Tegyük fel, hogy az előző fejezetben foglaltak csak „félig-meddig" győztek meg minket a szakértői rendszerek megbízhatóságáról, s így a szabályok következmény-részének bizonytalanságát minden esetében tekintsük mostantól *50%*-osnak, azaz `CF := 50` minden `rule/3` szabály minden `rhs/2` következményére! A tudásbázisban elhelyezkedő szabályok tehát mostantól a következők:

```
rule(rule_1, lhs([av(amerikai,kurtz),
            av(fegyver,m1),
            av(orszag,nono),
            av(ellenseges,nono),
            av(szemszin,kek),
            av(elad,kurtz-m1-nono)]),
          rhs(av(bunozo, kurtz),50)
    ).

rule(rule_2, lhs([av(amerikai,dr_strangelove),
            av(fegyver,m1),
            av(orszag,nono),
            av(ellenseges,nono),
            av(szemszin,fekete),
            av(elad,kurtz-m1-nono)]),
          rhs(av(bunozo, kurtz),50)
    ).
```

[12] *CF*, mint *Certainty Factor*

[13] Látható tehát, hogy csak $(p_1 \wedge p_2 \wedge \ldots \wedge p_n \rightarrow q)$ alakú szabályokkal, ún. *Horn-klózokkal* foglalkozunk, ahol $i=1,2,\ldots,n$-re p_i és q (negált vagy ponált) állítások.

```
rule(rule_3, lhs([av(ellensege,nono-amerika)]),
          rhs(av(ellenseges, nono),50)
    ).

rule(rule_4, lhs([av(birtokol,nono-m1),
            av(raketa,m1)]),
          rhs(av(elad,kurtz-m1-nono),50)
    ).

rule(rule_5, lhs([av(raketa,m1)]),
          rhs(av(fegyver,m1),50)
    ).
```

A fenti szabályok tehát rendre – az előző fejezetben tárgyalt – következő szabályokat valósítják meg:

```
bunozo(kurtz) :-
     amerikai(kurtz),
     fegyver(m1),
     orszag(nono),
     ellenseges(nono),
     szemszin(kek),
     elad(kurtz,m1,nono).

bunozo(dr_strangelove) :-
     amerikai(dr_strangelove),
     fegyver(m1),
     orszag(nono),
     ellenseges(nono),
     szemszin(fekete),
     elad(kurtz,m1,nono).

ellenseges(nono) :-
     ellensege(nono,amerika).

elad(kurtz,m1,nono) :-
     birtokol(nono,m1),
     raketa(m1).

fegyver(m1) :-
     raketa(m1).
```

A fenti szabályokon túl adott volt még egynéhány konkrét tény-adat is… Mivel most a tények is bizonytalanok (mint pl. a szabályok következményei), ezért mostantól a

tényeket is a munkamemóriában reprezentáljuk arra az esetre, hogyha netalán változtatni kellene rajtuk (pl. netalán megváltozik bizonytalanságuk, stb). A tények reprezentációjára a `fact/2` predikátumot használjuk. Ennek alakja a következő legyen:

```
fact(Fact,CF)
```

A `fact/2` predikátum első argumentuma (`Fact`) egy `av/2` alakú predikátum, második argumentuma (`CF`) pedig az adott tény ismeretének – eddig is látott – bizonytalansági mérőszáma (*%*-ban). A következő tényeket ismerjük (tfh., *50%*-os bizonyossággal):

```
fact(av(ellensege,nono-amerika),50).

fact(av(birtokol,nono-m1),50).
```

Ezek – az előző szakaszban tárgyalt – következő tényeknek felelnek meg:

```
ellensege(nono,amerika).

birtokol(nono,m1).
```

Látható, hogy mind a szabályok, mind az ismert tények reprezentálásánál a több-argumentumú predikátumokat úgy alakítottuk át attribútum-érték párokká, hogy a nevezett predikátumok nevét tekintettük az attribútum megnevezésének, míg argumentumait „kötőjelekkel” összekötve egyetlen értékének vettük. Például az `a(b,c,d)` predikátumot `av(a,b-c-d)` alakban reprezentáljuk, stb.

Miután elkészültünk a tudás – tudásbázisban és munkamemóriában való – reprezentálásával, elkezdhetjük kidolgozni a visszafelé láncolt következtetést megvalósító következtető motorunkat! A probléma jelen esetben ugyebár az, hogy *„Ki a bűnöző? Bűnöző-e X?…”*, stb. Erre eddig a `bunozo(X)` kérdést kellett a Prolog-nak feltennünk. Most azonban saját eljárást kell konstruálnunk erre a célra. Az eljárás – híven ahhoz, hogy *„X”* egy megoldását keressük – legyen egy `findgoal/2` nevezetű klóz! A `findgoal/2` eljárásnak a következőket kell tudnia:

- Már ismert a probléma attribútum-érték formában adott megoldása;
- Ha még nem ismert a probléma megoldása, akkor össze kell tudnia gyűjteni a visszafelé láncolt következtetésre alkalmas szabályokat;
- Tudnia kell, ha adott esetben a felhasználóhoz kell fordulnia valamilyen információ beszerzése céljából, melyet máshonnan nem tudhat meg.

A felhasználótól „kérdezhető” attribútumokat egy-egy `askable/2` predikátum tudásbázisban való elhelyezése jelezhetné a rendszernek. Ennek első argumentuma a „kérdezhető” attribútum megnevezése lenne, míg a második maga a kérdés, mint atom. Például: `askable(dagadt, 'Mennyire dagadt?').` Ezek szerint tehát a következő állításokkal bővül tudásbázisunk:

```
askable(szemszin,'Milyen a szemszin? ').

askable(amerikai,'Ki az amerikai? ').

askable(orszag,'Melyik az orszag? ').

askable(raketa,'Milyen a raketa? ').
```

Miután már rendelkezésünkre áll a szükséges „tudás”, valósítsuk meg a `findgoal/2` eljárás megfelelő klózait! Az első – ismert megoldásért felelős – klóz a következő:

```
findgoal(av(Attr,Val),CF) :-
     fact(av(Attr,Val),CF),
     !.
```

Az első `findgoal/2` klóz tehát azt az esetet fedi, ha a keresett – `av/2` predikátum formájában adott – bizonyítandó cél-állítás már adott a munkamemóriában. Látható, hogy a bizonyítandó cél-hipotézist az első argumentumban kell specifikálnunk… Ha tehát van már megoldás, akkor – a vágó hatására – a többi klózra már nem kerül sor. A második `findgoal/2` klóz a felhasználó megkérdezésének esetét fedi:

```
findgoal(av(Attr, Val),CF) :-
     \+ fact(av(Attr,_),_),
     askable(Attr,Prompt),
     query_user(Attr,Prompt),
     !,
     findgoal(av(Attr,Val),CF).
```

A második `findgoal/2` klóz tehát, abban az esetben, ha az adott, bizonyítandó cél még nem ismert, ám az attribútum „kérdezhető”, felteszi a `Prompt`-ban adott kérdést a felhasználónak, s a választ az adott attribútum értékeként, egy megfelelő `fact/2` állítás formájában eltárolja a munkamemóriában, azaz `assert`-álja (megjegyzi) a megismert tényt. A felhasználói interfészben implementálandó `query_user/2` predikátum tehát a következő:

```
query_user(Attr,Prompt) :-
     write(Prompt),
     read(Val),
     write('Milyen valoszinuseggel? '),
     read(CF),
     assert(fact(av(Attr,Val),CF)).
```

Látható, hogy a `query_user/2` klóz nem csak az `Attr` attribútum értékére kérdez rá, hanem arra is, hogy milyen valószínűséggel igaz az, hogy valóban ez az `Attr` attribútum értéke (*-100* = biztosan hamis; *100* = biztosan igaz). Már csak egy dolog van hátra a `findgoal/2` klózzal kapcsolatban: megalkotni a harmadik, visszafelé láncolt következtetésért felelős, leglényegesebb klózt! Bizonytalan esetben azonban figyelembe kell vennünk azt is, hogy különböző valószínűségekkel, de több szabály is alkalmas lehet arra, hogy egy-egy adott cél-hipotézisre (konkrétan egy-egy adott attribútum-érték párosra) vonatkozó állítást igazságát befolyásolja. Minden cél esetén tehát végig kell haladnunk az összes olyan szabályon, amely a célt, mint következményt tartalmazza. E szabályok valószínűségeit rendre össze kell kombinálnunk, hiszen ezek együttesen befolyásolják a végeredményben kapott, úgymond „bizonyított" hipotézis bizonyosságát. Ha ez a bizonyosság 100%, akkor kész vagyunk. Ellenkező esetben csak akkor állunk le, ha már nincs több – célra vonatkozó – szabály.

A szabályok kiválasztásával, kapcsolatos valószínűségek kombinálása igen egyszerűen történik. Legyen adott egy `LHS→RHS` alakú szabály, továbbá egy `RHS`-re vonatkozó `CF` bizonyosság! A szabály akkor „süthető el", avagy akkor „tüzelhető", ha a baloldal (`LHS`) igaz. A baloldal igazsága attól függ, hogy a benne szereplő állítások együttes bizonyossága elér-e egy minimális küszöbértéket (pl. *20%*). Ha nem, akkor e szabály alapján nem lehet következtetni a jobboldalra (`RHS`). Ha igen, akkor bizonyára lehet. A baloldal együttes bizonyosságát tekintsük a benne szereplő állítások bizonyossága minimumának! Ha tehát a baloldalban szereplő összes állítás bizonyossága ismert számunkra, azaz az állítások mind-mind ismert tényként szerepelnek a munkamemóriában, akkor kiszámítható az együttes bizonyosságuk. Ha létezik olyan állítás a szabály baloldalán, amely még nem ismert, akkor a szabály nem „süthető el".

Tegyük fel, hogy a fentebb említett szabály „tüzelhető". Ekkor meg kell vizsgálnunk, hogy a jobboldal (`RHS`) által definiált tény ismert-e már, azaz szerepel-e a munkamemóriában valamilyen `OldCF` bizonyossági faktor mellett. Ha még nem szerepel, akkor a szabály „tüzelésének" következtében már szerepelni fog. Bizonyossága ekkor a szabály premisszája és konklúziója bizonyosságának kombinációjaként adódik. Láttuk, hogy a premissza bizonyossága a benne szereplő állítások bizonyosságának minimuma[14]. Jelöljük ezt a bizonyosságot `Tally`-val, míg az adott szabály

[14] A premissza bizonyosságának meghatározása csupán megállapodás kérdése. Számunkra azonban ez az egyszerű megoldás is bőven elegendő.

konklúziójának bizonyosságát CF-fel! Ekkor a premissza és a konklúzió „kombinált” bizonyossága (NewCF) pl. a következőképp számolható:

```
NewCF = (CF*Tally)/100
```

Ha tehát az RHS-nek megfelelő tény még nem szerepel a munkamemóriában (avagy a megfelelő attribútum-érték páros még nem ismert), akkor tényként be kell szúrnunk (assert) a munkamemóriába NewCF bizonyossággal. Ha azonban a szabály jobboldalán (RHS) szereplő, bizonyítandó cél-állítás már szerepel egy adott (OldCF) bizonyossággal a munkamemóriában (azaz már valamilyen bizonyossággal ismert), akkor a helyzet egy kicsit bonyolultabb. Ekkor ugyanis a „tüzelt” szabály „kombinált” bizonyosságát (NewCF) össze kell „kombinálni” a munkamemóriában már szereplő cél-állítás bizonyosságával (OldCF). Ez a „kombinált” bizonyosság lesz a munkamemóriában elhelyezkedő cél-állítás „új” bizonyossága. Ennek az „új” bizonyosságnak a kiszámítására az MI körökben jól ismert – orvosi szakértői rendszerként úttörő jelentőségű – MYCIN klasszikus megoldását alkalmazzuk. Jelöljük ezt az „új” bizonyosságot CF(X,Y)-al:

$$CF(X,Y)=\begin{cases} X+\dfrac{Y(100-X)}{100}, X\geq 0, Y\geq 0 \\ X+\dfrac{Y}{1-\min(|X|,|Y|)}, X<0, Y>0 \\ X+\dfrac{Y}{1-\min(|X|,|Y|)}, X>0, Y<0 \\ -CF(-X,-Y), X<0, Y<0 \end{cases}$$

Látható, hogy X és Y helyébe rendre NewCF és OldCF kerül a mi megoldásunkban. Implementáljuk tehát a fent elmondott elveket – hozzuk létre a harmadik findgoal/2 klózt:

```
findgoal(Goal, CurCF) :-
     fg(Goal, CurCF).
```

A findgoal/2 harmadik klóza tehát egy fg/2 segédeljáráson alapszik. Az fg/2 eljárás klózai a következők:

```
fg(Goal, CurCF) :-
     rule(_N, lhs(IfList), rhs(Goal, CF)),
     prove(IfList, Tally),
     adjust(CF, Tally, NewCF),
     update(Goal, NewCF, CurCF),
     CurCF == 100,
     !.

fg(Goal, CF) :-
     fact(Goal, CF).
```

Az `fg/2` klóz működését már fentebb összefoglaltuk: egyrészt talál egy megfelelő, a célra illeszthető szabályt. Előállítja a szabály baloldalának együttes bizonyosságát (`Tally`), majd ezzel összekombinálja a szabály jobboldalának bizonyosságát (`CF`). A kapott bizonyosság (`NewCF`) alapján frissíti a szabály jobboldalának megfelelő tény (`Goal`) munkamemóriabeli bizonyosságát. Ha ez a bizonyosság (`CurCF`) *100%*-os, akkor a jobboldalnak megfelelő cél (`Goal`) bizonyítása sikerült, s így befejeződik. Ha a bizonyosság még kisebb *100%*-nál, akkor az első `fg/2` klóz meghiúsul, és a másodikra kerül a sor, amely mindenképp sikerül (Miért is? ☺), egyúttal meghatározva a cél-állítás aktuális – munkamemóriabeli – bizonyosságát.

Az `fg/2` eljárásban a `prove/2` eljárás felelős a premissza együttes valószínűségének kiszámításáért. Ennek implementációja a következő:

```
prove(IfList, Tally) :-
     prov(IfList, 100, Tally).

prov([], Tally, Tally).
prov([H|T], CurTal, Tally) :-
     findgoal(H, CF),
     min(CurTal, CF, Tal),
     Tal >= 20,
     prov(T, Tal, Tally).
```

Látható, hogy a `prove/2` klóz egy `prov/2` segédeljárásra támaszkodik. Első argumentuma (`IfList`) a vizsgált szabály baloldalán elhelyezkedő attribútum-érték párokkal reprezentált állítások listája. Második argumentuma (`Tally`) kimenő – evvel egyesítődik a lista által reprezentált állítások együttes bizonyossága.