EGE ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

İLERİ PROGRAMLAMA DİLLERİ

SUNUM RAPORU

MANTIK PROGRAMLAMA

HAZIRLAYAN:

Tahir Emre KALAYCI

TESLİM ALAN:

Doç. Dr. N.Yasemin TOPALOĞLU

17 KASIM 2003
Bornova, İZMİR

1. Mantık Nedir?

Mantık, bilgi ve altında yatan doğrulama işi arasındaki ilişkilere dayanır. Mantık programlamanın kaynağı "Predicate Calculus"tür, çünkü ispat yapabilmek için gereken net bir karakterize etme tekniği içermektedir. İspatlar formal yada sembolik olarak algoritmalarla ifade edilebilen, analiz edilebilen matematiksel nesnelerdir.

Gerçek yaşamdaki bazı meselelerin durumlarından bahsederken genellikle

Nazım resimi çok sever.
Abidin resim seven herkesin dostudur.

şeklindeki tanıtıcı (declarative) cümleler kurarız.
Bu cümleleri okuyan çoğu kişi hemen aşağıdaki sonuç cümlesini rahatlıkla üretebilecektir:

Abidin, Nazım'ın dostudur.

Bu şekilde insan düşünce sürecini aydınlatmamıza/biçimselleştirmemize yarayan sisteme mantık denmektedir. Yukarıdaki şekildeki mantık klasik mantık olup aşağıdaki gibi bu ifadeleri sembolik mantıkla ifade edebiliriz:

A1:sever(Nazım,resim)
A2:HER X sever(resim) → dost(Abidin,X)
T1:dost(Abidin,Nazım)

Doğal insan dili ilk görüşte ideal gibi gözükse de uzun,belirsiz,şüpheli olması ve içerik gerektirmesi nedeniyle onu ifade etmek için filozoflar ve matematikçiler bir şeyler aramışlardır.Mantık ta bunlardan biridir.

Imperative yaklaşım da denilen Von Neumann yöntemi makinenin işleyeceği yönergelerin özel mimarilerde aşağıdan yukarıya(bottom-up) olarak türetilir. Buna örnek olarak aşağıdaki doğal sayıların karesini üreten algoritmayı verebiliriz:

```
    Sayı=0;
Döngü:Kare=Sayı * Sayı;
    Yaz(Kare);
    Sayı=Sayı+1;
    Git Döngü;
```

Bu algoritmayı teknik ayrıntılara göre daha uygun hale getirmek için daha yüksek seviyeli programlama paradigması ile yeniden tanımlamamız daha iyi bir biçimselleştirme olacaktır. Burada oluşacak olan kritik soru mantığı nerede kullanacağımızdır, problemi temsil etmede mi, yoksa problemi çözmede mi?

Bunun için Mantık yardımıyla bu doğal sayıların karesini alma işleminin teknik ayrıntılarını yazalım:
(Kolaylık olsun diye "Peano" gösterimi kullanılmıştır)

Sayılar: 0→ 0 , 1→ s(0) , 2→ s(s(0)) , 3→ s(s(s(0)) , ….
Doğal sayıların tanımı: doğal(0) VE doğal(s(0)) VE doğal(s(s(0))) VE …

Daha iyi bir çözüm: doğal(0) VE HER X (doğal(X)→ doğal(s(X)))

Doğal Sayıların Toplanması:

HER X (doğal(X)→ topla(0,X,X)) VE

HER X HER Y HER Z (topla(X,Y,Z)→ topla(s(X),Y,s(Z)))

Doğal Sayıların Çarpılması:

HER X (doğal(X)→ çarp(0,X,0)) VE

HER X HER Y HER Z (çarp(X,Y,W) VE topla(W,Y,Z) → çarp(s(X),Y,Z))

Doğal Sayıların Karesi:

HER X HER Y (doğal(X) VE doğal(Y) VE çarp(X,X,Y)→ doğal_kare(X,Y))

En son yazdığımız ifade imperative programın teknik ayrıntısı olarak görülebilir.

Bu yaptığımız çözüm gösterim için iyidir. Ama efektif tümdengelim yordamı yardımıyla bu problemi çözebiliriz(Greene).

- Tümdengelim yöntemini bilgisayarda bir kez ve hepsini programla
- Problem için uygun bir gösterim bul
- Çözümleri elde etmek için sorular sor ve geri kalanı tümdengelim yordamına bırak.

Yukarıdaki tanımlamamızı aşağıdaki şekilde işleyebiliriz:
Sorgu: doğal(s(0))?
Yanıt: (evet)
Sorgu: HERHANGİBİR X topla(s(0),s(s(0)),X)?
Yanıt: X=s(s(s(0)))
Sorgu: HERHANGİBİR X topla(s(0),X,s(s(s(0))))?
Yanıt: X=s(s(0))
Sorgu: HERHANGİBİR X doğal(X)?
Yanıt: X=0 VEYA X=s(0) VEYA X=s(s(0)) VEYA ….
Sorgu: HERHANGİBİR X HERHANGİBİR Y topla(X,Y,s(0))?
Yanıt: (X=0 VE Y=s(0)) VEYA (X=s(0) VE Y=0)
Sorgu: HERHANGİBİR X doğal_kare(s(s(0)),X)?
Yanıt: X=s(s(s(s(0))))
Sorgu: HERHANGİBİR X doğal_kare(X,s(s(s(s(0)))))?
Yanıt: X=s(s(0))
Sorgu: HERHANGİBİR X HERHANGİBİR Y doğal_kare(X,Y)?
Yanıt: (X=0 VE Y=0) VEYA (X=s(0) VE Y=s(0)) VEYA (X=s(s(0)) VE Y=s(s(s(s(0))))) VEYA …..

Buraya kadar mantık ile problemleri nasıl gösterebileceğimizi ve ne şekilde çözebileceğimizi gördük, fakat bu işleri hangi mantık ile ve hangi yargılama(muhakeme:reasoning procedure) yordamı ile yapacağımıza aşağıdaki kriterlere göre karar vermemiz gerekir:

- Efektif bir yargılama yordamımız var mı?
- Anlatım gücümüzü yeterli mi?
- Mantığımızın altında yatan özellikleri ve teorik olarak limitlerimizi biliyor muyuz?

Seçim yapacağımız mantıklara aşağıdaki listedekiler örnek olabilir:

- Önermesel (propositional) mantık
- Birinci dereceden yükleme dayanan matematik (First order predicate calculus)
- Yüksek dereceli mantıklar
- Modal mantık
- λ Calculus

Seçim yapılacak muhakeme (reasoning) yordamlarına da şu liste:

- Doğal tümdengelim, klasik yöntemler
- Kesin çözüm (resolution)
- Prawitz/Bibel
- Bottom-up fixpoint
- Yeniden yazma (rewriting)
- Daraltma (narrowing)

2. Mantık Programlama

2.1. Tarihçe

Mantığı bir programlama dili olarak kullanma fikri bilgisayar bilimleri topluluğunda uzun zaman önceden beri var olan bir fikirdir. Erken zamanlarda teoremleri ispat eden programlar geliştirilmişti ama hız ve depolamaya bağlı kısıtlı kapasitede ve daha ciddi olan ve arama karmaşıklığından kaynaklanan problemler vardı. Alan Robinson tarafından 1960'lı yıllarda geliştirdiği, bir teoremin ispatlanmasında ispatın yaratılması için rehber olarak kullanılan Resolution ilkesi mantıksal ispatlar kurulmasına dayanan programlama dillerinin gerçekleştirilmesi hedefine büyük katkı sağlamıştır. Bu ilkeyi takiben Londra'daki Imperial Kolejdeki Robert Kowalski ve arkadaşlarının ve aynı dönemde Alain Colmerauer ve arkadaşlarının bir doğal dilin çözümlenmesi (parsing) için çalıştırılabilir gramerler (executable grammars) üzerine çalışmaları sonucunda Prolog dili 1972 yılında gerçekleştirildi. Üzerinden geçen 20 yıla karşın mantık programlama alanı çekiciliğini çoğu bilgisayar bilimcisi için kaybetmemiş ve yeni bilimciler için araştırma konusu olmuş faal durumunu kaybetmemiş olup araştırmalar halen sürmektedir.

Mantıksal programlamanın gelişimi sırasında geçilmiş önemli adımlar şunlardır:

- 1934'te Alfred Tarski "Predicate Calculus" için anlamsal bir teori geliştirmiştir.
- 1936'da Gerhard Gentzen "Bir cümle için bir Predicate Calculus ispatı varsa bu ispat konu dışı içeriklere girilmeden cümlenin içeriğinden de üretilir." fikrini ortaya atmıştır. Gentzen'in bulduğu sonuçlar "Predicate Calculus" cümlelerinin ispatlarını bulan algoritmaların arayışını hızlandırmıştır.
- İlk teorem ispatlayan algoritmaların yaratılması teşebbüsü ispatı mümkün olan algoritmalar arasında yorucu aramalara neden olmuştur. 1950'lerde üretilen bilgisayarların gücüne rağmen, öne sürülen "predicate calculus" ispat prosedürlerinin karmaşıklıkları çok zorlayıcı olmuştur.
- 1960'larda "predicate logic"in daha basit şekli olan "clausal logic" ortaya çıkmıştır. Bu, verimli hesaplanabilir prosedür üretmek için çok önemli bir ispattır. Çoğunlukla aynı şeyi ifade etmenin alternatif yolları olduğu için standart "predicate logic" gereksizdir. Çünkü iki ifade yazım olarak tamamen farklı olsa da anlamları aynı olabilir. Bu durum bilgiyi "predicate logic" olarak ifade etmek isteyen insanlar için bir avantaj iken ifadeleri işleyen prosedürleri verimsizleştirir. Daha basit "clausal logic", daha farklı ve daha etkin ispat işleme yaklaşımlarına yol açmıştır.

- 1963'te J. Alan Robinson "resolution" olarak adlandırdığı çok basit ve çok etkin bir çıkarım kuralını prensip alan bir "clausal logic" icat etmiştir. Sezgisel bir çıkarım kuralı olmamasına rağmen "resolution", bir goal'e ulaşmak için kullanılan ispat prosedürünün büyük adımlarla ilerlemesine izin vermektedir. "Resolution", "unification" denilen bir işleme dayanmaktadır. Jacques Herbrand tarafından öne sürülen bu işlem "clause" ifadelerinin direk olarak hesaplanmasına izin vermekte ve daha önceki teorem ispatçılarının yaptığı detaylı aramaları ortadan kaldırmaktadır.
- Mantığa dayalı yapay zeka sistemlerinin geliştirilmesine tanımlayıcı bir yaklaşım, mantıksal programlama araştırmalarının ilerlemesiyle aynı döneme rastlamaktadır. 1960' ların sonunda yapay zeka komiteleri, yapay zeka amaçları için bilginin tanımlanarak mı yoksa prosedürel olarak mı daha iyi gösterilebileceğini tartışıyordu.
- 1972'de Robert Boyer ve Strother Moore tarafından implementasyon stratejilerinin geliştirilmesi için çalışmalar yapılmış ve bu çalışmalar popular ve uygulanabilir bir mantıksal programlama dili olan Prolog'u ortaya çıkarmıştır.
- 1974'te Robert Kowalski bilginin "Horn Clause" gösterimi ile prosedürel gösteriminin denkliğini ispatlamıştır. Bu aşamadan sonra mantık programlama sistemlerinin uygulanması ve geliştirilmesi için teknolojik bir ortam oluşmuştur.

2.2. Mantık Programlamanın Temelleri

2.2.1. Mantık Programlama Nasıl?

Bir mantık programı axiom kümeleri ve bir hedef cümlesinden oluşur. Hedef cümlesinin doğruluğu için axiomların yeterli olup olmadığının tespiti için çıkarım kuralları vardır. Mantık programının çalışması axiomlardan hedef cümlesinin ispatının kurulmasına denktir. Bu dillerin diğer bir adı da kural tabanlı dilleridir. Bu isim belirli durumların sağlanması ve sağlandığı zaman buna uygun bir tepkinin verilmesinden verilmiştir. Bu dala giren en çok bilinen dil Prolog dilidir.

Mantık programlama sayesinde programlar hakkında muhakemeler yapmamız, çıkarımlar yapmamız sağlanır. Ayrıca bu işin otomatik olarak yapılabilmesi açısından kolaydır. Otomatik dönüşümlere, paralelizasyon, yarı otomatik debugging ve program doğrulama için kapı açar. Geliştirilmiş programlama üreticiliği vardır.

Mantık programlama modelinde programcı temel mantıksal ilişkileri tanımlamaktan sorumlu olup çıkarım kurallarının uygulanmasının stilinden sorumlu değildir. Böylece

Mantık+Kontrol=Algoritmalar olmaktadır.

Mantık programlama değişken kümelerine dayanmaktadır. Belirtimler (predicate) değişken kümelerinin veri tipinin soyutlanması ve genelleştirilmesidir. Bir değişken kümesini

S0xS1xS2x…..xSn

şeklinde ifade edebiliriz. Örneğin doğal sayıların kare alma fonksiyonu aşağıdaki şekilde değişken kümeleriyle ifade edilebilir:

{(0,0),(1,1),(2,4),..}

Bu şekildeki bir kümeye ilişki denir ve aşağıdaki ifadede kare alma ilişkisinin tanımı görülmektedir:

Doğal_kare={(0,0),(1,1),(2,4),..}

Her çiftler için genelleştirme ve isim için bir soyutlama yaparak aşağıdaki ifadeyi elde ederiz:

Doğal_kare=(x,x^2)

İsmi parametreleştiririz ve şu ifadeyi elde ederiz:

Doğal_kare(X,Y)<-- Y is } X*X

Burada çift kümesi Doğal_kare olarak adlandırıldı ve parametre olarak X,Y ifadelerini kullanıyoruz.

2.2.2. Mantık Sistemlerinin Sözdizimi (Syntax)

Mantık bir dil olduğu için her dil gibi bir sözdizimi vardır. Mantık sisteminin sözdizimi, sembollerden üretilen iyi biçimledirilmiş formül (well formed formulas-legal logical expressions)leri tanımlar. Tipik sözdizimi aşağıdaki elemanları içerir:

Sabitler (Constants)
Fonksiyonlar (Functions)
Yüklemler (Predicates)
Değişkenler (Variables)
Bağlaçlar (Connectives)
Tanımlayıcılar (Quantifiers)
Noktalama işaretleri

2.2.3. Mantık Sistemlerinin Semantiği

Kullanacağımız dilin anlambilimini (semantic) bilmeden o dilin ifade ettiklerini anlamayız. Bu yüzden mantıkta kullanılan formülleri anlamak için önce anlambilimini öğrenmeliyiz.

W bir formüller kümesi olsun (İçerdiği tüm formüllerin birleşimi) ve A bir formül olsun,

W | = A

İfadesi her W modeli için A doğrudur manasına gelir. Bunu açıklayacak olursak her W nin doğru olması durumu için, A da doğru olacaktır.Bu yüzden aşağıdaki şekilde cümleler ile bunu açıklarız:

W A yı gerektirir (W entails A)
W A yı içerir (W implies A)
A W den sonra gelir (A follows from W)
A W nin mantıksal sonucudur (A is a logical consequense of W)
A W nin bir teoremidir (A is a theorem of W)

Örnek: Aşağıdaki yüklemlerle (predicates) temsil edilen bloklar dünyamız olduğunu varsayalım:

üstünde (A, B): A B nin üstündedir
yukarısında (A, B): A B nin yukarısındadır
W kümemiz aşağıdaki formülleri içersin (a, b, c farklı blokları temsil eden sabitler)
üstünde (a, b)
üstünde (a, b)

HER X, Y için yukarısında(X, Y) ← üstünde(X, Y)
HER X, Y için yukarısında(X, Z) ← üstünde(X, Y) & yukarısında(Y, Z)

Bu verilere göre yukarısında (a, c) ifadesinin doğru olduğunu görürüz. yukarısında (c, b) nin de yanlış olduğunu. Ama bu veriler olmadan bunu söyleyemezdik. Bu yüzden bu tip yargılara varabilmek için mantık verilerini oluşturmamız gerektirmektedir.

Önermesel(Propositional) Mantığın Anlambilimi : Önermesel mantık önermeler ve bağlaçlardan oluşur. Tipik bağlaçlar şunlardır:

Λ (&,VE),V (VEYA), ¬ (DEĞİL),← (→olarakta yazılabilen "implication")

Bu bağlaçların herbiri için bir doğruluk tablosu (şekildeki) kullanılıp içinde bulundukları cümlelerin sonuçları üretilir.

A	B	A Λ B	A V B	¬A	A ← B

0	0	|	0	0	1	1
0	1	|	0	1	1	0
1	0	|	0	1	0	1
1	1	|	1	1	0	1

Önermesel mantık için, verilen sonlu sayıda eleman içeren bir küme için sonlu sayıda atama olabileceği için bu doğruluk tablosundan her zaman yararlanılır. Çok fazla sayıda önerme içeren ifadeler için bu tablo çok büyük olacaktır.

Örnek: W=(a← (a V b))

D= a V b

W | = D ?

Aşağıdaki tabloyu kullanarak bu soru için HAYIR cevabını verebiliriz:

a	b	|	a ← (a V b)	a V b
0	0	|	1	0
0	1	|	0	1
1	0	|	1	1
1	1	|	1	1

Bağlaçların durumuna bir formül ya Doğru yada Yanlış olacaktır. Formülü Doğru durumunda değerlendiren yorumlamaya formülün modeli denilir. Bir formül en az bir model içeriyorsa tatmin edici (*satisfiable*) , eğer her yorumlama için doğru ise totoloji ve her yorumlama için yanlış ise "contradiction" adını alır.

Mantık Sistemlerinin Çıkarım Kuralları(Inference Rules): W | = a kullanarak iyi biçimlendirilmiş formüller arasındaki ilişkiler hakkında konuşabiliriz. Önermesel mantık için kullandığımız doğruluk tablosunu "predicate" mantık(ileride bahsediliyor) için kullanamayacağımızdan dolayı, bunun yerine çıkarım sistemi kullanırız. Çıkarım sistemi yeni formüller türetmemize yarayan çıkarım kurallarından oluşan bir sistemdir. Örneğin "modus-ponus" yaygın kullanılan bir çıkarım kuralıdır:

Ne zaman bir x ve x→y e sahipsek y yi türetebiliriz

$$\frac{x,\ x \rightarrow y}{y}$$

İspat, iyi biçimlendirilmiş formüllerin bir serisidir:a1,a2,a3,a4,….,aN şeklinde herbir ai W kümesinden yada aj<ai şeklinde aj lerden oluşan bir kümeden elde edilen bir çıkarımdır.

W | = D

İfadesini W kümesinden D formülünü türetebildiğimizi göstermek için kullanırız. Türetmeler yalnızca çıkarım kurallarıyla belirlenir. Elbette çıkarım sistemimizin doğru çalışmasını ve herhangi bir mantık sonucunu kaçırmasını istemeyiz. Çıkarım kuralları W den D ye bir ispat üretemiyorsa, W | = D doğru değilse "sound" olurlar. Eğer W | = D doğru ise ve W den D ye bir ispat üretebiliyorsa "complete" çıkarım sistemimiz vardır.

2.2.4. "Predicate Logic"

"Clausal Form": "Implication"(→) ve eşitlik(equivalence) operatörleri yardımıyla yazılan "Predicate Calculus" cümleleri Λ (conjunction), V (disjunction) ve ¬ (negation) operatörleri kullanılarak ta yapılabilirler. Bütün "Predicate Calculus" cümleleri "clausal form" adı verilen özel bir biçimde ifade edilebilir. "Clausal form" "Predicate Calculus" cümleleri Prolog cümlelerine benzetilebilir. Bu nedenle "clausal form" Prolog ve mantık arasındaki ilişkiyi anlamak açısından önemlidir.

Bir "Predicate Calculus" cümlesinin normal forma çevrilmesi altı ana adımda gerçekleşir:

1. Adım: "Implication"ların yokedilmesi : İlk adımla → ve ↔ operatörleri ortadan kaldırılır.

a→B	¬a∨B
a↔B	(a∧B)∨(¬a∧¬B)
a↔B	(a→B)∧(B→a)

Bu durumda HER(X,adam(X)→insan(X)) cümlesi HER(X, ¬adam(X)∨insan(X)) olarak çevrilebilir.

2. Adım : "Negation" operatörünü içe doğru hareket ettirmek

Eğer "negation" operatörü atomik olmayan bir cümleye uygulanırsa bu adım uygulanır.

¬(insan(sezar) ∧ yaşıyor(sezar)) cümlesi

¬(insan(sezar) ∨ ¬ yaşıyor (sezar))

olarak çevrilebilir.

Başka bir örnek vermek gerekirse ¬HER(Y,şahıs(Y)) cümlesi VAR(Y,¬şahıs(Y)) olarak çevrilebilir.

Kurallamak gerekirse

¬(a∧B)	¬a∨¬B
¬VAR(v,P)	HER(v,¬P)
¬HER(v,P)	VAR(v,¬P)

3. Adım : Skolem normal form

Diğer bir adım varoluş tanımlayıcılarını(existential quantifier) ortadan kaldırmaktır. Bu işlem varoluş değişkenler yerine "Skolem" sabit olarak adlandırılmış yeni sabit semboller kullanılarak gerçekleştirilir. Bu özelliklerde en az bir nesne vardır demek yerine böyle bir nesne için bir isim yaratılır ve bu özelliklere sahip olduğu söylenir. Bu ifadenin anlamını en çok bozan çevrim "skolem normal form"a çevrimdir.

Örnek vermek gerekirse

VAR(X,kadın(X) ∧ annesi(X,havva))

"Skolem normal form"a çevrilirse

$$kadın(g197) \wedge annesi(g197,havva)$$

Burada g197 başka hiçbir yerde kullanılmamış yeni bir sabittir. Sabit g197 annesi havva olan bir kadını gösterir. Daha önce kullanılmış bir sembol seçmek önemlidir, çünkü

$$VAR(\ X, kadın(X) \wedge annesi(X,havva))$$

demek belirli bir kişinin HAVVA'nın kızı olduğu anlamına gelmez, sadece böyle bir insanın varlığını bildirir.

Cümle içinde evrensel tanımlayıcı(universal quantifier) varsa cümleyi "skolem normal form"a çevirmek daha zorlaşır. Örnek olarak

$$HER(X,insan(X)\rightarrow VAR(Y,annesi(X,Y)))$$

"her insanın annesi vardır" cümlesi

$$HER(X,insan(X)\rightarrow annesi(X,g2))$$

olarak çevrilirse cümle "bütün insanların annesinin g2 sabitinin gösterdiği aynı kişi olduğu anlamına gelir. Cümle içinde evrensel tanımlayıcı ile kullanılmış değişkenler varsa, "skolem normal form"a çevrilirken sabit yerine fonksiyon sembolü kullanılır. Ve biraz önceki cümlenin doğru olarak çevrilmesi aşağıda gösterilmiştir:

$$HER(X,insan(X)\rightarrow annesi(X,g2(X)))$$

Bu durumda g2 fonksiyon sembolü sonuç olarak verilen herhangi bir kişinin annesini döndürür.

4. Adım : Evrensel tanımlayıcıları dışa doğru hareket ettirmek

Bütün evrensel tanımlayıcılar cümlenin dışına doğru hareket ettirilir. Bu adım anlamı bozmaz. Örnek olarak:

$$HER(X,kadın(X) \wedge HER(Y,çiçek(Y)\rightarrow sever(X,Y)))$$

cümlesi

$$HER(X,HER(Y,kadın(X) \wedge (çiçek(Y)\rightarrow sever(X,Y)))$$

şeklinde çevrilir.

Evrensel tanımlayıcı cümlenin dışına alındığında ekstra bilgi sağlamaz. Bu durumda evrensel tanımlayıcılar cümleden çıkartılabilirler. Sonuç olarak yukarıdaki cümle

$$kadın(X) \wedge çiçek(Y)\rightarrow sever(X,Y)$$

olarak çevrilebilir.

5. Adım : $\wedge$ bağlacını $\vee$ bağlaçları üzerine dağıtmak.

Bu adıma gelindiğinde "Predicate Calculus" cümlesinin çok değiştiği görülmektedir. Artık cümledeki tanımlayıcılar yok olmuştur. Bağlaç olarak ise sadece $\wedge$ ve $\vee$ bağlaçları kalmıştır. Bu adımda cümle yeni bir forma ("conjunctive normal form") sokulacaktır ve artık cümlede "conjunction"lar "disjunction"lar içinde yer almayacaklardır.

Çevrim aşağıdaki kurallar doğrultusunda gerçekleştirilecektir.

$(A \wedge B) \vee C$ işlemi $(A \vee C) \wedge (B \vee C)$

$(A \vee B) \wedge C$ işlemi $(A \wedge C) \vee (B \wedge C)$

olarak çevrilecektir.

6. Adım : "clause" forma sokmak

Yukarıdaki formül dikkatle incelenirse $\wedge$ bağlaçlarıyla bağlanmış literal veya literallerden oluştuğu görülür. Literal grupları $\vee$ bağlacıyla bağlanmış literallerdir.

$\vee$ bağlacıyla bağlanmış literal gruplarının ayrıntılarına inilmezse cümlenin genel gösterimi

$(A \wedge B) \wedge (C \wedge (D \wedge E))$

şeklindedir. Parantezler gereksizdir çünkü aşağıdaki cümlelerin hepsi aynı anlama gelmektedir.

$(A \wedge B) \wedge (C \wedge (D \wedge E))$

$(A \wedge ((B \wedge C) \wedge (D \wedge E))$

$(A \wedge B) \wedge ((C \wedge D) \wedge E)$

Kısacası yukardaki cümle

$A \wedge B \wedge C \wedge D \wedge E$

şeklinde yazılabilir.

Son olarak literaller arasında $\wedge$ olduğunu belirtilmesine gerek yoktur, çünkü literaller arasındaki bağlaçların hepsi aynıdır. Bu durumda cümle literaller cümlesi olarak görülebilir:

$\{A,B,C,D,E\}$

Oluşan küme içerisindeki literallerin her biri "clause" olarak adlandırılır. Bunun sonucunda bir "Predicate Calculus" cümlesinin bir "clause"lar kümesi olduğu söylenebilir.

Bu "clause"lar ayrıntılı olarak incelenirse her birinin bir literal veya $\vee$ bağlacıyla bağlı literaller grubu olduğu görülür.

$((W \vee X) \vee Y) \vee (Z \vee Q)$

"clause" içinde literaller arası bağlaçlar hep aynı olduğu için bir "clause" aşağıdaki şekilde gösterilebilir.

$\{W,X,Y,Z,Q\}$

Orjinal cümle "clausal form"a dönüştürüldü. Özet olarak "clausal form"daki bir cümle her biri bir literal kümesi olan "clause"'lardan oluşmuş bir kümedir. Bir literal atomik bir cümle veya bir atomik cümlenin tersidir.

Bu 6 adımı gösteren bir örnek:

$HER(X,HER(Y,şahıs(Y) \rightarrow saygı\_duyar(Y,X)) \rightarrow kral(X))$

Bu cümle “Eğer herkes bir kişiye saygı duyuyorsa bu kişi kraldır” anlamına gelmektedir.

1. adım sonunda:

HER(X, ¬(HER(Y, ¬şahıs(Y) ∨ saygı_duyar(Y,X))) ∨kral(X))

2. adım sonunda:

HER(X, VAR(Y,şahıs(Y) ∧ ¬(saygı_duyar(Y,X)) ∨ kral (X))

3. adım sonunda:

HER(X,şahıs(f1(X)) ∧ ¬saygı_duyar(f1(X),X)) ∨ kral (X))

Not: f1 bir skolem fonksiyondur.

4. adım sonunda:

(şahıs(f1(X)) ∧¬saygı_duyar(f1(X),X)) ∨ kral(X)

5. adım sonunda (conjunctive normal form):

(şahıs(f1(X)) ∨ kral(X)) ∧ (¬saygı_duyar(f1(X),X) ∨ kral(X))

Burada iki “clause” vardır. Birinci “clause” iki literalden oluşmuştur

şahıs(f1(X)) kral(X)

İkinci “clause”daki literaller ise

¬saygı_duyar(f1(X),X) kral(X)

“Clause”lar İçin Bir Gösterim:“Clausal” form adı yazılmış bir cümlenin bir “clause”lar kümesi olduğu bilinmektedir. Şimdiki gösterimde “clause”lar ardarda yazılmaktadır. Bir “clause” terslenmiş ve terslenmemiş literallerden oluşur. Bu iki grup “:-“ işaretiyle ayrılır. Terslenmemiş literaller noktalı virgüller(;), terslenmiş literallerse tersleme işaretleri olmadan virgülle(,) ayrılmış olarak yazılırlar.

Örnek olarak P1;P2;.....,PM :- Q1, Q2, ..., QN.

Bu gösterim Prolog dilindeki “clause”larla aynıdır. Bu gösterim üzerinde biraz işlem yapılırsa gösterimin ne anlama geldiği daha iyi anlaşılacaktır.

(P1∨P2∨P3∨...∨PM) ∨(¬Q1∨¬Q2∨¬Q3∨...∨¬QN)

≡(P1∨P2∨...∨PM) ∨¬(Q1∧Q2∧...∧QN)

≡(Q1∧Q2∧...∧QN) →(P1∨P2∨...∨PM)

Bu durumda Prolog dilindeki “,” “∧” sembolüne, “;” “∨” sembolüne, “:-“ “→” sembolüne karşılık gelir.

Bu durumda

HER(X,HER(Y,şahıs(Y) → saygı_duyar(Y,X)) → kral (X))

≡(şahıs(f1(X)) ∨ kral(X)) ∧ (¬saygı_duyar(f1(X),X)

cümlesi Prolog’da aşağıdaki gibi yazılabilir :

sahis(f1(X));kral(X):-.

kral(X):-saygı_duyar(f1(X),X).

2.2.5. Resolution Yöntemi

Şimdiye kadar anlatılan kısımda "Predicate Calculus" cümleler düzenli bir forma sokulmuştur, bu bölümde eldekilerin ne işe yaradığı anlatılacaktır. Eldeki verilerden yola çıkarak bu verilerin mantıksal sonuçları (logical consequences) çıkarılmaya çalışılacaktır. Bu işlem açıkça görüldüğü gibi teorem ispatıdır.

1960'lı yıllarda insanları makinların teoremleri otomatik olarak ispatlayacak şekilde programlamak istediler. J.Alan Robinson, "resolution principle" tekniğini ortaya attı. Resolution principle mekanik teorem ispatıdır. Resolution "clausal form"daki cümlelerle çalışır. Birbiriyle ilişkili iki "clause"dan, bu "clause"ların mantıksal sonucu olan yeni bir "clause" yaratır.

Resolution tekniğindeki temel prensip şudur:

Aynı literal bir "clause"un sol tarafında ve diğer "clause"un sağ tarafında bulunursa, iki "clause" biraraya getirilip tekrar eden literal atılarak yeni bir "clause" oluşturulur.Örnek olarak;

üzgün(turgay);kızgın(turgay):-çalışma_günü(bugün),yağmurlu(bugün).

tatsız(turgay) :- kızgın(turgay), yorgun(turgay).

Resolution tekniği ile

üzgün(turgay); tatsız(turgay):-çalışma_günü(bugün),yağmurlu(bugün),yorgun(turgay).

cümlesi çıkartılabilir.

Yukarıdaki örnekte iki şey atlanmıştır. Birincisi "clause"lar değişken içerdiğinde durum daha da zorlaşır. İkincisi tekrar eden literaller tamamıyla aynı olmak zorunda değildir. İki literal birbiriyle aynı olmasa da eşlenebilir.

İkinci örnekte bu iki durumun nasıl çözüme kavuşturulduğu anlatılmıştır:

1. şahıs (f1(X));kral(X):-.
2. kral(Y):-saygı_duyar(f1(Y),Y).
3. saygı_duyar(Z,arthur):-şahıs(Z).

(2) ve (3) numaralı "clause"lardan aşağıdaki sonuç çıkarılabilir.(saygı_duyar literallerini eşleyerek).

4. kral(arthur):-şahıs(f1(arthur)).

Y ile arthur, Z ile f1(Y) eşlendi. Bir sonraki adımda (1) ve (4) numaralı "clause"lardan "Arthur bir kraldır" sonucunu çıkaracağız.

5. kral(arthur);kral(arthur):-.

Resolution tekniğinde eşleme(matching) olarak adlandırılan işlem "unification" adını alır.

Resolution tekniği belirli bir şeyi ispatlamak için nasıl kullanılır? Resolution adımları uygulanmaya devam edilir. Yukarıdaki örnekte "kral(arthur)" "clause"unu mantıksal bir sonuç olsa da verilen "clause"lardan çıkarmak mümkün değildir.

Bu durumda resolution tekniğinin güçlü bir teknik olmadığı gibi bir sorunla karşılaşılır. Neyse ki resolution tekniği eğer "clause" mantıksal bir sonuçsa problemi çözmeyi garanti eder.

Resolution tekniğinin bu önemli özelliği "refutation complete" olması olarak bilinir.

Bu teknikle ispatlamaya çalışılan "clause"un terslenmiş hali ("goal") hipotezlere eklenir ve resolution tekniği uygulanır. Sonuçta "empty clause" oluşursa ispatlanmaya çalışılan teorem doğrudur.

Bu prensip yukarıdaki tekniğe uygulanırsa;

6.:-kral(arthur). (hedefin değillenmesi (negation of goal))

5.kral(arthur);kral(arthur):-.

7.kral(arthur):-.

(6) ve (7) numaralı "clause"lardan aşağıdaki sonuç elde edilir:

:-. (boş clause)

Bu sonuç bir "Arthur bir kraldır" "clause"unun (1),(2) ve (3) numaralı "clause"ların mantıksal sonucu olduğunu göstermiştir.

2.3. Mantık Programlamanın Alanları

Mantık programlama kökleri AI'e dayanan güçlü bir metafordur. Bir mantık programı spesifikasyon olarak deklaratif okumaya sahiptir, bir teori mantık kümeleri şeklinde ifade edilmektedir. Mantık programlamanın özellikle kullanıldığı problem alanları aşağıdaki gibidir:

- Teorem kanıtlama
- Yorumlarıcı ve derleyiciler
- Çizge algoritmaları
- Protokol doğrulama
- Bilgi temsili
- Uzman sistemler, karar destek sistemleri
- Planlama ve programlama (scheduling)
- Doğal dil işleme
- Veritabanı modelleme

Bu alanlarda aşağıdaki uygulamalar geliştirilmektedir:

- İlişkisel veritabanları
- Doğal dil arayüzleri
- Uzman sistemler
- Sembolik denklem çözümleri
- Planlama
- Prototip üretimi
- Simülasyon
- Programlama dil gerçekleştirimleri

Alternatif programlama şemaları ise şu şekildedir:

- coroutining
- concurrency
- Denklemlere dayalı mantık programlama
    - Fonksiyonel yetenekler
- Sınırlamalı mantık programlama (constraint)
    - Mantıksal aritmetik, kümeler, …
- Tümdengelimli veritabanları

3. Prolog

3.1. Genel Bakış

Prolog en geniş şekilde kullanılan mantık programlama dilidir. Bazı özellikleri:

- Mantıksal değişkenleri kullanır.Bunlar diğer dillerdeki değişkenlere benzemezler. Programcı bunları veri yapıları içerisindeki hesaplama sürdükçe doldurulabilen "delikler" olarak kullanabilir.
- Birleştirme (Unification) parametreleri geçiren, sonuçları döndüren, veri yapılarını seçen ve yaratan kurulu manipulasyondur.
- Temel kontrol akış modeli backtracingtir.
- Program cümleleri ve veri aynı forma sahiptir.
- Yordamların ilişkisel formu 'tersinir'(reversible) yordamlar tanımlanabilmesine olanak sağlar.
- Cümleler durum analizi ve nondeterminizmi ifade etmek için elverişli bir yol sunar
- Bazı durumlarda mantığın içermediği kontrol özellikleri kullanılabilir.
- Bir prolog programı kuralları içerdiği gibi olgularıda içeren bir ilişkisel veritabanı olarak gözükebilir.

İfadeler bir problem hakkında çözümün nasıl yapılacağı yerine neyin doğru olduğuna dair bilgiler içerir. Prolog sistemi ifadeleri mümkün çözümler uzayı içerisinden bir çözüm arama yolu ile çözümü gerçekleştiren ifadeler içerir. Bütün problemler saf deklaratif tanımlamalar içermediğinden bazen "extralogical" cümleler gerekebilir.
Prolog için gramer örneği:

Core Prolog

Program ::= Clause... Query | Query
Clause ::= Predicate . | Predicate :- PredicateList .
PredicateList ::= Predicate | PredicateList , Predicate
Predicate ::= Atom | Atom(TermList)
TermList ::= Term | TermList , Term
Term ::= Numeral | Atom | Variable | Structure
Structure ::= Atom (TermList)
Query ::= ?- PredicateList .
Numeral ::= an integer or real number
Atom ::= string of characters beginning with a lowercase letter or encluded in apostrophes.
Variable ::= string of characters beginning with an uppercase letter or underscore

Terminals = {Numeral, Atom, Variable, :-, ?-, comma, period, left and right parentheses }

Bir Prolog programı örneği:

```
elephant (george).
elephant (mary).

panda (chi_chi).
panda (ming_ming).
```

```
dangerous(X) :- big_teeth(X).
dangerous(X) :- venomous(X).

guess (X, tiger) :- stripey(X), big_teeth(X), isaCat(X).
guess (X, koala) :- arboreal(X), sleepy(X).
guess (X, zebra) :- stripey(X), isaHorse(X).
```

3.2. Prolog Temelleri

Cümleler: Gerçekler ve Kurallar

Bir Prolog programcısı nesneleri, bağıntıları (relation) ve bu bağıntıların doğru olduğu kuralları tanımlar. Örnek olarak

Bill likes dogs.

Bu örnek, "bill" ve "dog" nesneleri arasındaki bağıntıyı gösterir. Bu cümleye bağıntının ne zaman doğru olduğunu gösteren bir kural eklenirse:

Bill likes dogs if the dogs are nice.

Gerçekler: Bilinenler

Prolog'da nesneler arası bağıntılar "predicate" olarak adlandırılır.

Aşağıda konuşma dilinde "likes" bağıntısı örneklenmiştir:

Bill likes Cindy.

Cindy likes Bill.

Bill likes dogs.

Takip eden örneklerde ise Prolog dilinde yazılmış bazı gerçekler görülmektedir;

```
likes(bill, cindy).

likes(cindy, bill).

likes(bill, dogs).
```

Gerçekler nesnelerin özelliklerini belirtmek amacıyla da kullanılırlar:

"Kermit is green."	green(kermit)
"Caitlin is a girl"	girl(caitlin)

gibi.

Kurallar: Verilen Gerçeklerle Hangi Sonuçlara Ulaşılabilir

Kurallar verilen gerçeklerden yeni gerçekler türetmeye yararlar. Aşağıda "likes" bağıntısıyla ilgili bazı kurallar listelenmiştir:

Cindy likes everything that Bill likes.

Caitlin likes everything that is green.

Verilen kurallar doğrultusunda aşağıdaki yargılara ulaşılabilir:

Cindy likes Cindy.

Caitlin likes Kermit.

Aynı kuralları Prolog dilinde yazmak gerekirse:

likes(cindy,Something):-likes(bill,Something).

likes(caitlin,Something):-green(Something).

Sorgular (“Queries”)

Prolog’da bir dizi gerçek verildiğinde; bu gerçeklerle ilgili sorular sorulabilir ve bu işleme Prolog sistemini sorgulamak denir. Verilen gerçekler ve kurallar doğrultusunda aşağıdaki sorular cevaplanabilir:

Does Bill likes Cindy?

Prolog dilinde bu soru aşağıdaki gibi sorulur:

likes(bill, cindy).

Bu sorguya karşılık Prolog “yes” cevabını döndürür.

Sorulabilecek ikinci bir soru “What does Bill likes?” sorusudur. Bu soru Prolog diline aşağıdaki gibi çevrilebilir:

likes(bill,What).

Dikkat edilirse Prolog dilinde gerçekler ve sorgular arasında yazım farkı yoktur. Burada dikkat edilmesi gereken nokta birinci nesne (“bill”) küçük harfle başlar. Bu “bill”in sabit bir nesne olduğunu gösterir. “What” ise büyük harfle başlarve değişken bir nesnedir.

Prolog bir sorguya cevap verebilmek için daima en tepeden başlayarak gerçekleri tarar. Prolog tarama işlemine taranacak gerçek kalmayana kadar devam eder. Yukarıdaki sorgunun sonucu aşağıdaki gibi döndürülecektir:

What=cindy

What=dogs

2 solutions

Sorguyu değiştirip “What does Cindy likes?” şekline getirirsek:

likes(cindy, What).

Prolog aşağıdaki sonucu döndürecektir:

What=bill

What=cindy

What=dogs

3 solutions

Gerçekler, Kurallar ve Sorguların Birleştirilmesi

Aşağıdaki gibi gerçeklerin ve kuralların verildiği varsayılırsa:

A fast car is fun.

A big car is nice.

A little car is practical.

Bill likes a car if the car is fun.

Bu gerçekler incelendiğinde Bill'in hızlı arabalardan hoşlandığı sonucu çıkartılabilir. Prolog'da aynı sonucu bulur. Hızlı arabaların eğlenceli olduğu bilgisi verilmese bile bir insan hızlı bir arabanın eğlenceli olacağı sonucunu tahmin edebilirdi. Fakat Prolog'un tahmin edebilme yeteneği yoktur; sadece verilen gerçekleri bilir.

Aşağıda Prolog'un sorguları cevaplamak için gerçekleri nasıl kullandığını gösteren bir örnek sunulmuştur. Program-I'in aşağıda verilen parçasındaki gerçekler ve kurallar incelenirse :

```
likes(ellen, tennis).
likes(john, football).
likes(tom, baseball).
likes(eric, swimming).
likes(mark, tennis).
likes(bill, Activity):-likes(tom, Activity).
```

Programın son satırı bir kuraldır:

```
likes(bill, Activity):-likes(tom, Activity).
```

Bu kural konuşma dilinde aşağıdaki cümleye karşılık gelir:

Bill likes an activity if Tom likes that activity.

Bu kuralda likes(bill,Activity) baş kısım; likes(tom,Activity) ise gövde kısmıdır. Bu örnekte Bill'in "baseball"dan hoşlandığı hakkında bir bilgi yoktur. Prolog'un cevabı bulması için aşağıdaki sorgu verilebilir:

```
likes(bill,baseball).
```

Bu sorgunun cevabını bulabilmek için Prolog aşağıdaki kuralı kullanır:

```
likes(bill, Activity):-likes(tom, Activity).
```

Program bütün olarak yazılırsa:

```
PREDICATES
        nondeterm likes(symbol, symbol)
CLAUSES
        likes(ellen, tennis).
```

likes(john, football).
likes(tom, baseball).
likes(eric, swimming).
likes(mark, tennis).
likes(bill, Activity):-likes(tom, Activity).

GOAL

likes(bill,baseball).

Sistem diyalog penceresine aşağıdaki sonucu döndürür:

Yes

Sistem likes(tom,baseball) gerçeği ile likes(bill, Activity):-likes(tom, Activity) kuralını birleştirip likes(bill,baseball) gerçeğine ulaşmıştır.

Aşağıdaki sorguya cevap bulmak istenirse:

likes(bill,tennis).

Sistem "no" cevabını gönderir çünkü program parçasında Bill'in tenisten hoşlandığı gerçeği verilmediği gibi verilen gerçeklerden ve kurallardan da bu sonuç çıkarılamamaktadır.

Değişkenler: Genel Cümleler

Prolog'da değişkenler kullanılarak genel gerçekler ve kurallar yazılabildiği gibi genel sorularda sorulabilmektedir. Konuşma dilinde değişkenler çok sık kullanılır. Örnek olarak aşağıdaki İngilizce cümle ele alınırsa:

Bill likes the same thing as Kim.

cümle Prolog'da

likes(bill,Thing):-likes(kim,Thing).

şeklinde yazılır. Yukarıda görüldüğü gibi Prolog'da değişkenler büyük harfle veya "_" karakteri ile başlar.

3.3. Prolog'un Çalışma Prensipleri

Unification

Prolog bir goal'ü bir subgoal (call) ile uyuşturmaya çalışırken Unification adı verilen bir arama işlemi yapılır. Unification, subgoal'de bulunan veri yapıları ile verilen clause'daki bilgileri uyuşturma çabasıdır. Unification, diğer geleneksel dillerde bulunan parametre iletme, durum seçimi (case selection), yapı yaratma (structure building), yapı erişimi (structure access) ve atama gibi işlemleri implement eder. Aşağıdaki örnek program incelenecek olursa:

```
DOMAINS
        title, author = symbol
        pages = unsigned
PREDICATES
        book(title, pages)
        nondeterm written_by(author, title)
        nondeterm long_novel(title)
CLAUSES
        written_by(fleming, “DR NO”).
        written_by(melville, “MOBY DICK”).
        book(“MOBY DICK”, 250).
        book(“DR NO”, 310).
        long_novel(Title):- written_by( _ , Title),
                            book(Title, Length),
                            Length > 300.
```

Verilen örnek program için goal:

```
written_by(X,Y)’dir.
```

Bu goal’ü sağlayacak veriyi bulmak için Prolog, programa verilmiş tüm “written_by” clause’larını test eder. X ve Y değişkenlerini “written_by” clause’undaki veri yapıları ile eşleştirmek için program baştan sona doğru taranır. Bu goal ile uyuşan bir clause bulunduğunda clause’un değerleri X ve Y değişkenlerine bağlanır, böylece goal ile clause identical (özdeş) hale gelirler. Yani goal ile clause unify edilmiş (birleştirilmiş) olur. Bu, Unification işlemidir.

Bu programda goal, ilk “written_by” clause’u ile özdeşleşebilir.

```
written_by( X , Y ).
written_by(fleming,”DR NO”).
```

X değişkeni *fleming* verisine, Y değişkeni ise “DR NO” verisine bağlanır. Program çalıştırıldığında sonuç:

```
X=fleming, Y=DR NO
```

olarak görüntülenir.

Prolog tüm çözümleri aradığından programın goal’ü varsa diğer “written_by” clause’ları ile de özdeşleşir:

```
written_by(melville,”MOBY DICK”).
```

Ve Prolog ikinci çözümü de görüntüler:

```
X=fleming, Y=DR NO
X=melville, Y=MOBY DICK
2 solutions.
```

Eğer programa "written_by(X, "MOBY DICK")" goal'ü verilirse Prolog yine tüm "written_by" clause'ları ile goal'ü uyuşturmaya çalışacaktır. Karşılaşacağı ilk clause'da "MOBY DICK" ve "DR NO" verileri uyuşmadığından program ikinci gerçek (fact) olan

written_by(melville,"MOBY DICK")

clause'unu deneyecektir. Bu clause, goal ile özdeşleşeceği için X değişkeni *melville* verisi ile bağlanır.

Örnek program için verilen bir diğer goal

long_novel(X)'tir.

Prolog bu goal'ü gerçekleştiren X değişkenine değer atarken clause'lar arasındaki "long_novel(Title)" kuralını (rule) bulur. Title da X gibi bir değişkendir ve ikisi de clause'a bağlanmış durumda değildir (unvounded). Ama goal ile Clauses bölümünde verilmiş olan

```
long_novel(Title):- written_by( _ , Title),
                    book(Title, Length),
                    Length > 300.
```

kuralı uyuşmakta olduğu için unification yapılır. Prolog sırasıyla kuralın alt goal'lerini sağlamaya çalışır. Önce kuralın içindeki ilk subgoal olan

written_by(*, Title) ele alınır. Burada kitabın yazarının kim olduğu bilgisi önemli olmadığından Author değişkeni yerine "*_" sembolü konmuştur. Prolog önce ele aldığı ilk subgoal için bir çözüm arar. Yani yazarın adı önemsenmeksizin "written_by" clause'una uygun Title verileri aranır.

```
written_by( _ , Title ).
written_by(fleming,"DR NO").
```

İlk veri için Title değişkeninin değeri "DR NO" olur. Daha sonra bir sonraki subgoal olan

book(Title, Length)

ele alınır. Bu clause'a uygun verinin aranması sırasında daha önceden Title değişkenine bağlanmış olan "DR NO" bilisi için uygun olan Length bilgisi aranmaktadır. Yani asıl çözümü aranan clause artık

book("DR NO", Length)

olmuştur. Programın ilk bulacağı veri

book("MOBY DICK", 250)

olacağı için burada bir uyuşma sağlanamayacaktır. Dolayısıyla program bir diğer "book" clause'una bakacaktır. Diğer clause'da "DR NO" verisi uyuştuğu için Length değişkeninin değeri 310 olarak belirlenecek, 310 verisi ile Length değişkeni birbirine bağlanacaktır. Bu noktadan sonra, asıl goal olan "long_novel" kuralının üçüncü goal'ü olan

Length > 300

subgoal olur ve Length değişkenine bağlanan 310 verisi için subgoal sağlanmış olur. Çünkü Length değişkenine bağlanmış sayının 300'den büyük olması uzun roman olmak için aranan üçüncü koşulu da sağlamaktadır. Bu durumda "long_novel" kuralının üç koşulunu da sağlayan veri "DR NO" olur ve program çıktısı aşağıdaki gibi elde edilir.

X = DR NO

Backtracking

Bir problemin çözümünü ararken mantıksal bir yol izlemek gereklidir. Ama bazen izlenen mantıksal yol istenen sonuca ulaşamayabilir. Böyle bir durumla karşılaşıldığında başka alternatif çözüm yolları denemek gerekir. Örneğin bir labirentin içinde çıkış yolu bulmak üzere dolaşılırken izlenebilecek ve kesin çözüme götürecek yöntemlerden biri de yolun her ikiye ya da üçe ayrıldığı noktada hep soldaki yolu seçmektir. Bu şekilde ilerlenirken çıkmaz bir yere gelinirse en son sola dönülen yere kadar geriye dönülerek bu sefer sağdaki yol seçilip yine bir sonraki yol ayrımlarından hep sola dönme yöntemi izlenebilir. Bu şekilde olası tüm yollar denenmiş olur ve sonuca mutlaka ulaşılır.

Prolog da bu yöntemin aynısını kullanmaktadır. Bu yönteme "Backtracking", yani "Geriye dön tekrar dene" yöntemi denmektedir. Prolog bir probleme çözüm ararken, iki olası durum arasında karar vermek zorunda kalabilir. Her seçim yaptığı noktaya bir "Backtracking point"), yani bir belirteç koyarak subgoal'lerden ilkini sağlama işini gerçekleştirmeye çalışır. Eğer seçtiği subgoal çözüme ulaşmazsa Prolog geriye dönecek, belirteç koyduğu noktada diğer bir alternatifi seçerek çözümü aramaya devam edecektir. Backtracking kavramı örnek bir program üzerinde açıklanabilir.

```
PREDICATES
        nondeterm likes(symbol,symbol)
        nondeterm tastes(symbol,symbol)
        nondeterm food(symbol)
CLAUSES
        likes(bill, X): - food(X), tastes(X,good).
        tastes(pizza,good).
        tastes(spinach,bad).
        food(spinach).
        food(pizza).
GOAL
        likes(bill,What).
```

Bu örnekteki kuraldan Bill'in tadı güzel olan yiyeceklerden hoşlandığı anlaşılmaktadır. Program Bill'in neden hoşlandığını aramak üzere

```
likes(bill,What)
```

goal'ünü gerçekleştirecektir. Bu iş için program baştan sona doğru taranmaya başlanır. Bulunan ilk clause'da What değişkeni X değişkeni ile *unify* edilir çünkü aranan ve bulunan clause'lar birbiriyle uyuşmuştur (match).

```
likes(bill, What)
likes(bill, X)
```

Unification işleminden sonra Prolog, ele aldığı kuralın goal'ü sağlayıp sağlamadığını araştırır. Bunun için kuralın ilk bölümü olan

```
food(X)
```

subgoal'ü ele alınır. Bu ilk subgoal'ü sağlamak için Prolog tekrar programın en başına döner. İlk clause'da X değişkeni *spinach* bilgisi ile sağlanır. Elde edilen çözüm dışında bir başka alternatif daha olduğu için

Prolog ilk çözümü bulduğu noktaya bir Backtracking point koyar. Bu nokta Prolog'un *spinach* verisi için çözüm gerçekleşmezse geriye döneceği ve diğer alternatifi deneyeceği noktadır. Bu veri ile Prolog

tastes(spinach, good)

subgoal'ünü sağlamaya çalışır. Bu subgoal kuralın ikinci maddesidir. Clause'lar arasında

tastes(spinach, good)

durumu sağlanmadığından Prolog Backtracking point koyduğu noktaya

food(spinach)

durumuna geri döner. Bu noktada Prolog başka bir değişkeni deneyebilmek için bu noktadan sonra bağladığı tüm değişkenleri bırakır. Programın bu noktada "food(X)" clause'u için deneyebileceği diğer alternatif veri *pizza*'dır. X değişkeni *pizza* verisine bağlanır. Bu noktadan sonra program sağlaması gereken ikinci subgoal olan

tastes(pizza,good)

clause'unu ele alır. Bu clause'a uygun bir durum olup olmadığı programın yine en başına dönülerek program sonuna kadar aranır. Aranan yeni subgoal program içinde bulunduğundan, çözümü aranan goal için başarıya ulaşılmış olur. Kontrol edilen

likes(bill,What)

goal'ü için "pizza" çözümü bulunur çünkü What değişkeni "likes" kuralındaki X değişkeni ile unify edilmiş ve X değişkeni de *pizza* verisine bağlanmıştır. Sonuç aşağıdaki gibi rapor edilir:

What = pizza

1 solution.

Prolog problemin birden fazla çözümü olduğu durumlarda sadece ilk çözümünü bulmakla kalmaz, tüm olası çözümleri de bulur. Aşağıda bir örnek program verilmiştir.

```
DOMAINS
        child = symbol
        age = integer
PREDICATES
        nondeterm player(child,age)
CLAUSES
        player(peter,9).
        player(paul,10).
        player(chris,9).
        player(susan,9).
```

Bu programın amacı olarak tenis klübünün 9 yaşındaki üyeleri arasında bir turnuva düzenlenmesi durumu aşağıdaki goal ile sağlanabilir:

```
player(Person1,9),
player(Person2,9),
```

Person1 <> Person2.

Prolog önce kuralın ilk clause'u olan

player(Person1,9)

subgoal'üne bir çözüm bulmayı dener. İlk subgoal Person1 değişkeninin *peter* verisi ile uyuşmasıyla sağlanmış olur. İlk subgoal'ü elde ettikten sonra ikinci subgoal olan

player(Person2,9)

clause'una çözüm aranır. İkinci subgoal için de *peter* çözümü elde edilir. Üçüncü subgoal'de hem Person1 hem de Person2 *peter* verisini çözüm olarak aldığı için son subgoal başarısız olmuş olur.bu yüzden Prolog başarılı olduğu son subgoal'e döner ve ikinci subgoal'deki Person2 değişkeni için başka bir çözüm arar.

player(Person2,9)

clause'u için bulunan bir sonraki veri *chris* verisidir. Buradan tekrar son subgoal'e dönülür.

Person1 <> Person2

subgoal'ü için elde edilen *peter* ve *chris* verileri birbirinden farklı olduğundan çözüm olarak turnuvada *peter* ile *chris*'in eşleşerek maç yapabileceği belirlenmiş olur.

Prolog tüm olası sonuçlara ulaşmayı hedeflediğinden tekrar bir önceki subgoal'e döner ve yeni çözümler araştırır. Bu subgoal için Person2 değişkeni *susan* bilgisi ile de eşleşebileceğinden Prolog son subgoal'ü tekrar dener. *peter* ve *susan* verileri farklı olduğundan probleme bir farklı çözüm daha bulunmuş olur: *peter* ve *susan*.

Başka sonuçlar bulmak üzere program tekrar ikinci subgoal'e döner ama artık ikinci subgoal'ü sağlayan farklı bir veri kalmadığından Backtracking işlemine ilk subgoal'e dönerek devam eder.

İlk subgoal'ün Person1 değişkeni için *peter* verisinden farklı bir veri olarak *chris* verisine ulaşmasından sonra ikinci subgoal tekrar ele alınır. Program ikinci subgoal'e alternatif çözüm olarak tekrar *chris* verisine ulaşır. Person1 ve Person2 değişkenlerine bağlanan veriler aynı olduğundan üçüncü subgoal sağlanamaz ve tekrar ikinci subgoal'e dönülür. İkinci subgoal için bir diğer alternatif veri *susan* olduğundan Person2 değişkenine *susan* verisi bağlanır. Bu durumda Person1 ile Person2 değişkenlerinin verileri farklı olduğu için turnuvada maç yapacak bir çift daha bulunmuş olur: *chris* ve *susan*.

Program bu şekilde ilerleyerek olası tüm sonuçları bulur ve problemin çözümü şu şekilde oluşur:

Person1=peter, Person2=chris

Person1=peter, Person2=susan

Person1= chris, Person2= peter

Person1= chris, Person2= susan

Person1= susan, Person2= peter

Person1= susan, Person2=chris

6 solutions.

Dikkat edilmesi gereken bir nokta Prolog'un gereksiz sonuçlar da bulabileceğidir. Çünkü Prolog peter verisini Person1'in çözümü olarak ya da Person2'nin çözümü olarak bulduğuyla ilgilenmez.

Backtracking'in Dört Temel İlkesi

- Subgoal'ler yukarıdan aşağıya doğru sırayla sağlanmalıdır.

- Clause'lar programın içinde bulundukları sıra ile test edilirler.
- Bir subgoal bir kuralın sol taraf değeri ile uyuştuğu zaman, kuralın sağ tarafının da doğruluğu sağlanmalıdır. Kuralın sağ tarafı doğruluğu sağlanacak bir alt goal'ler kümesidir.
- Bir goal, bulunması gereken tüm gerçekler bulunduğunda sağlanmış olur.

Çözümler İçin Arama Kontrolü

Prolog'un Backtracking kurma mekanizması gereksiz aramalara neden olabilir ve bu durum etkinsizliği arttırabilir. Örneğin verilen problemin tek çözümlerinin bulunması gereken zamanlar olabilir. Diğer durumlarda özel bir amaç tanımlanmış olsa bile ek çözümler için Prolog'u zorlamak gerekli olabilir. Buna benzer durumlarda Backtracking süreci kontrol edilmelidir.

Prolog, Backtracking mekanizmasını kontrol edecek iki araca izin verecek şekilde geliştirilmiştir.bunlardan ilki *fail* predicate, ikincisi ise *cut*'tır. *Fail* mekanizması programı Backtracking yapmaya zorlarken *Cut* Backtracking'i engellemek amacıyla kullanılır.

Fail Predicate Kullanımı

Prolog fail, yani başarısızlık durumunda backtracking'e başlar. Bazı durumlarda alternatif çözümler elde etmek için programı backtracking yapmaya zorlamak gereklidir.prolog *fail* adı verilen özel bir predicate'ı destekler. Bu komut programda başarısızlığa ulaşmayı ve programın backtracking yapmasını sağlar. *Fail* etkisi 2=3 gibi doğru olmayan karşılaştırmalar ve mümkün olmayan amaçları gerçekleştirmeyi sağlar. Bu komutun kullanımıyla ilgili bir örnek program aşağıda verilmiştir.

```
DOMAINS
        Name=symbol
    PREDICATES
        nondeterm father(name,name)
        everybody
    CLAUSES
        father(leonard,katherine).
        father(carl,jason).
        father(carl,marilyn).
        everybody :- father(X,Y),
                     write(X,"is",Y,"'s father. \n"),
                     fail.
        everybody.
```

Everybody predicate'ı bir kere başarılı bir biçimde sonlandığında Prolog programına backtracking yapmasını söyleyen herhangi bir koşul yoktur.bu yüzden father clause'u sadece bir çözümle programdan dönecektir. Ama *everybody* predicate'ı tüm çözümlerin bulunması için backtracking yapmak üzere *fail* predicate'ını içermektedir. Dolayısıyla program geri dönüp tekrar çözüm aramak suretiyle tüm olası sonuçları bulacaktır. Everybody predicate'ının amacı programın çalışmasından daha net bir cevap elde etmektir.

```
    GOAL
```

father(X,Y).

X=leonard, Y=katherine

X=carl, Y=jason

X=carl, Y=marilyn

ve

GOAL

everybody.

leonard is katherine's father.

carl is jason's father.

carl is marilyn's father.

goal ve çözümleri için aşağıdaki çıkarımlar yapılabilir:

- *Everybody* predicate'ı father(X,Y) clause'u için daha fazla sonuç elde etmek üzere backtracking mekanizmasını kullanır. Program *everybody* kuralına doğru backtracking'e zorlanır:

  father(X,Y), write(X,"is",Y,"'s father. \n"), fail.

- *Fail* durumu hiçbir zaman sağlanamadığı için (hep başarısızlık olduğu için) Prolog backtracking'e zorlanır. Program birden fazla sonuç oluşturabilen son subgoal'e kadar backtracking yapar. Böyle bir call (çağırma) non-deterministic olarak nitelendirilebilir. Non-deterministic call, deterministic olan ve sadece bir çözüm üretebilen call ile çelişmektedir.

- Write cümlesi tekrar sağlanamadığı için (yeni bir çözüm önerilmediği için) Prolog bu sefer kuralın ilk subgoal'üne backtracking yapar.

- Bir kuralın içinde geçen *fail*'den sonraya bir subgoal yerleştirilmesi gereksizdir. *Fail* her zaman başarısızlığa uğrayacağı için *fail*'den sonraya yerleştirilmiş bir subgoal'e erişmek mümkün olmayacaktır.

Cut – Backtracking'i Önlemek

Prolog, backtracking'i önlemek için kullanılan ve ünlem işareti (!) olarak yazılan bir *Cut* mekanizmasını içerir. *Cut*'ın etkisi basittir: bir *Cut*'ın üzerinden geçerek backtracking yapmak imkansızdır.

Cut programın içerisine bir subgoal'ün bir kuralın içerisineyerleştirildiği gibi yerleştirilir. İşlem sırası *Cut*'a gelince *Cut* çağrısı (call) başarı ile sonuçlanır ve eğer varsa bir sonraki subgoal çağrılır. *Cut* bir kere geçildiğinde bir daha *Cut*'tan önce ele alınmış subgoal'lere ve *Cut*'ın içinde bulunduğu predicate'ı tanımlayan diğer predicate'lere backtracking yapmak da mümkün değildir. *Cut*'ın iki ana kullanımı vardır:

1. Bazı olasılıkların asla anlamlı çözümlere ulaşmayacağı biliniyorsa alternatif çözüm aramak zaman kaybıdır. Eğer böyle bir durumda *Cut* kullanılıyorsa program daha hızlı çalışacak ve daha az bellek kullanacaktır. Bu durum Green *Cut* (Yeşil *Cut*) olarak adlandırılır.
2. Program mantığı *Cut*'ı gerektiriyorsa alternatif subgoal'lerin dikkate alınmasını önlemek için kullanılır. Bu da Red *Cut* (Kırmızı *Cut*) olarak adlandırılır.

Cut Nasıl Kullanılır?

Kuralın içindeki bir önceki subgoal'e backtracking'i önleme:

r1:- a,b,!,c.

Bu, Prolog programına a ve b subgoal'leri için bir çözümün yeterli olduğunu söyleme yoludur. Prolog backtracking yöntemini kullanarak c kuralı için farklı çözümler bulabilecekse de a ve b için alternatif çözümler bulmak üzere backtracking yapmasına izin verilmez. Ayrıca r1 kuralını tanımlayan diğer bir clause'a da backtracking yapılamaz.

```
PREDICATES
        buy_car(symbol, symbol)
        nondeterm car(symbol, symbol, integer)
        colors(symbol, symbol)
CLAUSES
        buy_car(Model, Color):- car(Model, Color, Price),
                                colors(Color, navy), !,
                                Price < 25000.
        car(maserati, green, 25000).
        car(corvette, black, 24000).
        car(corvette, blue, 26000).
        car(porsche, blue, 24000).
        colors(blue, navy).
        colors(black, mean).
        colors(green, preppy).
```

Bu örnek programda amaç deniz renginde ve bedeli 25000'den az olan bir Corvette araba bulmaktır. Buy_cay kuralındaki *Cut*, veritabanında sadece bir tane deniz renkli Corvette olduğu için, ve onun da bedeli istenenden yüksekse başka bir alternatif aramaya gerek olmadığı için konmuştur.

```
buy_car(corvette,Y)
```

goal'ü için car subgoal'ü çağrılır.

- İlk araba Maserati olduğundan test başarısız olur.
- Daha sonraki car clause'ları test edilir ve Color değişkenine black verisinin karşılık geldiği bir Corvette araba bulunur.
- Bir sonraki çağrıda seçilen arabanın navy rengi olup olmadığı test edilir. Programda black verisi navy bir renk olarak belirtilmediğinden test başarısız olur.
- Program bir kez daha car clause'a backtracking yapar ve navy kriterine uygun başka bir Corvette arar.
- Uygun araba bulunur ve rengi test edilir. Bu sefer renk navy olarak bulunduğu için bir sonraki subgoal, yani *Cut* işletilir. *Cut* hemen başarılı olur ve car clause'u için yapılmış değişken bağlanmaları sabitlenir.
- Sıra kuralın son işletilecek subgoal'üne gelir.
    - Price < 25000

- Bu test başarısız olur ve Prolog başka bir araba bulup test etmek için backtracking yapmaya teşebbüs eder. *Cut* backtracking'i önlediği için ve son subgoal için başka çözüm de olmadığından goal başarısızlıkla sonuçlanır. Yani aranan kritere uygun araba bulunamamıştır.

Bir sonraki clause'da backtracking'i önleme:

Cut, Prolog'a belli bir predicate için doğru clause'un seçildiğini anlatmak için kullanılabilir.

```
r(1):- !, a, b, c.

r(2):- !, d.

r(3):- !, c.

r(_):- write("This is a catchall clause.").
```

Bu örnek program kodunda *Cut*, r predicate'ını deterministik yapmaktadır. Burada Prolog r'yi tek bir integer argüman ile çağırır. Çağrının r(1) olduğu varsayılırsa:

- Prolog yapılan bir çağrıya uyan bir match yakalamak için programı araştırır ve r'yi tanımlayan ilk clause'u bulur. Çağrı için birden fazla olası çözüm olduğundan Prolog bu clause'a bir backtracking point yerleştirir.
- Daha sonra program kuralın maddelerini işlemeye başlar. İlk önce *Cut*'ı geçer ve böylece başka bir r clause'una backtracking yapma olasılığını ortadan kaldırmış olur. Bu durum mevcut backtracking point'leri elimine eder ve çalışma zamanı verimliliği arttırılır. Ayrıca hata-kapanı (error-trapping) olan clause'un "başka hiçbir koşulda r'ye yapılan çağrıya match etmediğinde" çalıştırıldığını garanti eder.

Bu yapı diğer programlama dillerinde bulunan "case" yapısı gibidir. Ayrıca kuralların en başına birer test koşulu da kodlandığına dikkat edilmelidir.

```
r(X):- X=1, !, a, b, c.

r(X):- X=2, !, d.

r(X):- X=3, !, c.

r(_):- write("This is a catchall clause.").
```

Yukarıda verilen örnekte olduğu gibi test koşulu mümkün olduğunca kuralın başına yerleştirilmelidir. Bu, programa verimlilik katar ve programı daha kolay okunur hale getirir. Örnek program:

```
PREDICATES
        friend(symbol, symbol)
        girl(symbol)
        likes(symbol, symbol)
CLAUSES
        friend(bill, jane):- girl(jane), likes(bill,jane), !.
        friend(bill, jim):- likes(ji,baseball), !.
        friend(bill, sue):- girl(sue).
        girl(mary).
        girl(jane).
```

```
girl(sue).
likes(jim,baseball).
likes(bill,sue).
```

Bu programda çözümü aranacak goal

```
friend(bill,Who) sorgusudur.
```

Programdaki *Cut*'lar olmadan Prolog iki çözüme ulaşacaktır:

Bill hem Jane'in hem Sue'nun arkadaşıdır. Ama friend'i tanımlayan ilk clause'daki *Cut* programa, bu clause sağlanırsa (Bill'in bir arkadaşı bulunursa) diğer arkadaşları aramak için devam etmenin gerekmediğini anlatmaktadır.

Clause'ların içinde bir çağrı (call) sağlanmaya çalışılırken backtracking yer alabilir. Ama bir çözüm bulunduğunda Prolog bir *Cut* geçer ve böylece yazılmış friend clause'ları Bill'in sadece bir arkadaşını çözüm olarak geri döndürür.

Determinizm (Karar Verme) ve Cut

Friend predicate'i *Cut*'lar olmadan kodlanmış olsaydı backtracking yaparak birden fazla çözüm üretmeye muktedir non-deterministic bir predicate olacaktı.

Prolog implementasyonlarında programcılar non-deterministic clause'lara çalışma zamanında yapılan bellek talepleri yüzünden özel bir dikkat göstermelidirler. Prolog non-deterministic clause'lar için içsel kontroller yapmakta ve programcının yükünü azaltmaktadır.

Hata ayıklama ya da diğer amaçlar için kullanıcının programa müdahale etmesi gerekebilir. Bunun için programcıya derleyici tarafından ***check_determ*** komutu sağlanmıştır. Eğer check_determ komutu programın çok başında eklenirse, Prolog derleme sırasında bir non-deterministic clause ile karşılaştığında uyarı verecektir.

Bir predicate'ı tanımlayan kuralın içerisine *Cut* eklemek sureti ile non-deterministic bir clause deterministic yapılabilir.

Örneğin friend predicate'ını tanımlayan clause'a *Cut*'lar yerleştirmek predicate'ın deterministic olmasına yol açar çünkü *Cut*'lar ile friend sadece ve sadece bir çözüm üretecektir.

Not Predicate

Not predicate'ı bir örnek program üzerinde açıklanabilir.

```
DOMAINS
        name=symbol
        gpa=real
PREDICATES
        nondeterm honor_student(name)
        nondeterm student(name)
        probation(name)
CLAUSES
```

```
honor_student(Name):- student(Name,GPA),
                      GPA>=3.5,
                      not(probation(Name)).
student("Betty Blue",3.5).
student("David Smith",2.0).
student("John Johnson",3.7).
probation("Betty Blue").
probation("David Smith").
```

Bu programın goal'ü GPA'i en az 3.5 olan ve deneme sürecinde bulunmayan şeref öğrencilerini bulmaktır.

Not kullanırken: *Not* yüklemi subgoal'ün doğruluğu kanıtlamadığı zaman başarıya ulaşır. Bu da, bağlanmamış değişkenlerin *Not*'ın içinde bağlanmasını önleyen bir durumla sonuçlanır. Boşta değişkenleri olan bir subgoal *Not* ile çağrılırsa Prolog bir hata mesajı döndürecektir.

"Free variables not allowed in 'not' or 'retractall'."

Bu durum gerçekleşir çünkü Prolog için bir subgoal'de bulunan boştaki bir değişkenin bağlanabilmesi için subgoal başka bir clause ile unify ve subgoal başarıya ulaşmış olmalıdır. Bağlanmamış değişkenlerle *Not* subgoal'ünde başetmenin yolu anonymous (anonim) değişkenler kullanmaktır. Clause'ların doğru ve yanlış yazımlarına aşağıda örnekler verilmiştir.

```
likes(bill, Anyone):- likes(sue,Anyone), not(hates(bill,Anyone)).
```

Bu örnekte Prolog, "hates(bill,Anyone)" ifadesinin doğru olamdığını farketmeden önce Anyone değişkenini likes(sue,Anyone) ifadesi ile bağlar. Bu clause sorun çıkarmadan çalışması gerektiği gibi çalışacaktır.

Bu kural *Not* önce kullanılacak şekilde tekrar yazılırsa baştaki değişkenlerin *Not* ile bağlanamayacağını belirten bir hata mesajı ile karşılaşılır.

```
likes(bill, Anyone):- not(hates(bill,Anyone)), likes(sue,Anyone).
```

Not(hates(bill,Anyone)) ifadesindeki Anyone değişkeni clause hata vermeyecek biçimde başka bir anonim değişkenle birleştirilse bile hala yanlış sonuç döndürecektir.

```
likes(bill, Anyone):- not(hates(bill,_)), likes(sue,Anyone).
```

Bu clause'da Bill'in Sue'nun hoşlandıklarından hoşlandığı ve Bill'in nefret ettiği birşeyin olmaması durumu aranmaktadır.

Not kullanımı çok dikkatli bir biçimde yapılmalıdır.yanlış kullanımı bir hata mesajı alınması veya program mantığında bir hata ile sonuçlanabilir.

4. Sonuçlar

Mantıksal programlama kendisini pekçok kullanım alanında kanıtlamış bir yöntemdir. Bu kullanım alanları arasında diagnostic uzman sistemler, doğal dil işleme, agent tabanlı kontrol sistemleri yer almaktadır. Bunlardan doğal dil işleme mantıksal programlamanın kullanıldığı en zor uygulama alanları arasında gösterilmektedir. Doğal dil işleme mantıksal programlama ile doğal dillerin gramer tabanlı

gösterimlerinin birleştirilmesiyle konuşma dilini anlayan araçalrın geliştirilmesi için çalışan bir uygulama alanıdır.

Mantıksal programlamanın Web'e açılması da bir diğer kullanım alanı olarak yerini almış bulunmaktadır. Mantıksal programlamanın kullanımındaki amaç Web için geliştirilen uygulamalara knowledge tabanlı tekniklerin uygulanabilirliğini sağlamaktır. Böyle bir yaklaşımın avantajları kaynaklar hakkında knowledge seviyesinde reasoning yapma ve bilgi filtrasyonu tekniklerine mantık tabanlı teknikler ekleyebilme olacaktır.

Prolog gibi mantıksal programlama için kullanılan bir programlama dilinin uygulama alanlarının genişletilmesini sağlayan bir etken de Prolog'un Java implementasyonlarının geliştirilmesi olmuştur. Böylece mantıksal programlamanın Java'nın etkisiyle Web üzerinden uygulamalarının da mümkün olabileceği düşünülmüştür.

Bazı agent uygulamalarında mantıksal programlama dillerinin sağladığı yüksek formalizasyon seviyesinden de faydalanılabilmektedir. Şu anda kullanılan agent'ların ilk mantık tabanlı modelleri ile agentların diğer programlama dilleri ile realize edilmeleri arasında bir geçiş mevcut değildir. Bu geçiş deklaratif bir mantık tabanlı programlama dilinin yazılım agentlarının realize edilmesi için kullanılmasıyla gerçekleştirilebilir.

Mantıksal programlamanın bir diğer kullanım alanı constraint programlamadır. Bu, programlama açısından çok esnek bir paradigma olup uygulamayı içeren constraint'i ve hesaplamaları yapan inference engine'i açıklar. Kontrol, scheduling ve bankacılık gibi pekçok kullanım alanı vardır.

Inductive mantıksal programlama ise mantıksal programlamanın önerdiği sunumlar arasında veri madenciliği ve makinelerin öğrenmesi süreçlerini inceleyen bir uygulama alanıdır..

4.1. Klasik ve Mantıksal Programlama Karşılaştırılması

Prolog'da bir problem gerçekleri (facts) belirten terimler ve kurallar (rules) olarak tanımlanır ve programın sonuca ulaşması sağlanır.Pascal, C, BASIC gibi diller prosedürel dillerdir; yani bilgisayara çözüme ulaşmak için geçeceği yollar adım adım fonksiyon ya da subroutine'ler yazılarak belirtilir.

4.2. Kural (Rule) ve Gerçeklerin (Facts) Prosedürlerle İlişkisi

Bir Prolog programında kural tanımı bir prosedür tanımı gibi düşünülebilir. Mesela

likes(bill,Something):- likes(cindy,Something)

kuralı Bill'in Cindy'nin hoşlandıklarından hoşlanacağını belirtir.bu mantıktan hareketle

say_hello:- write("Hello"), nl. ve greet:- write("Hello"),nl.

gibi prosedürlerin diğer programlama dillerindeki subroutine ya da fonksiyonlara karşılık geldiği söylenebilir.

Prolog'daki gerçekleri de prosedürler gibi düşünmek mümkündür:

likes(bill,pasta).

gerçeği Bill'in pasta sevdiği gerçeğini belirten bir ifadedir. Eğer Who ve What gibi iki değişkenle

likes(Who,What)

gibi bir sorgulama yapılırsa bu değişkenler bill ve pasta verileri ile sırasıyla bağlanacaktır.

4.3. Kuralları "Case Statement" Gibi Kullanma

Prolog kuralları ile diğer dillerin prosedürleri arasındaki büyük bir fark Prolog'un aynı prosedür için birden fazla alternatif tanım verilmesine izin vermesidir.

Pascal'daki "case" deyimini kullanıyor gibi birden fazla tanım Prolog'da da kullanılabilir. Prolog aradığını bulana kadar tanımlanmış tüm kuralları deneyecektir.

```
PREDICATES
        nondeterm action(integer)
CLAUSES
        action(1):- nl, write("You typed 1."), nl.
        action(2):- nl, write("You typed two."), nl.
        action(3):- nl, write("Three was what you typed."), nl.
        action(N):- nl, N<>1, N<>2, N<>3,
                        write("I don't know that number").
GOAL
        Write("Type a number from 1 to 3: "),
        readint(Num),
        action(Num).
```

Eğer kullanıcı 1, 2 veya 3 yazarsa action kuralı çağrılacak ve argümanı uygun değerlerle bağlanacak ve ilk üç kuraldan sadece biri ile uyuşacaktır.

"I don't know that number." Yazısının girilen sayının verilen aralık dışında olduğu durumlar dışında ekrana gelmemesine dikkat edilmelidir.bu mesajı yazabilmek için Prolog'un girilecek bir X sayısının 1, 2 ya da 3 olmadığını ispatlaması gereklidir (X<>1, X<>2 ve X<>3 testi). Bu subgoal'lerden biri başarısız olursa Prolog başka alternatifler arayacağından ve başka alternatifler de olmadığından clause'un devamı hiç çalıştırılmayacaktır. Bağlanmamış bir değişken bir sayıya eşit olamayacağından son kuralda hata mesajı alınacaktır.

4.4. Cut'ın GoTo Gibi Kullanılması

Yukarıda verilen örnek programda doğru kuralın seçilip çalıştırılmasından sonra Prolog diğer alternatifleri de tarayacağı için programda fazladan çalıştırılan bölümler olacaktır.

Eğer Prolog'a alternatifler aramayı bırakması söylenirse zaman ve bellekten tasarruf edilecektir. *Cut* bu programda:

İstenilen kuraldan geriye backtracking yapma ve başka alternatifleri arama anlamına gelecek şekilde kullanılabilir. Backtracking yapmak ise hala mümkündür. Çalışan kural başka bir kural tarafından çağrılıyor ve çağıran kuralın alternatifleri varsa bu alternatifler hala denenebilir. *Cut* kullanarak aynı program şöyle yazılabilir:

```
PREDICATES
        nondeterm action(integer)
CLAUSES
        action(1):- !, nl, write("You typed 1.").
```

```
        action(2):- !, nl, write("You typed two.").
        action(3):- !, nl, write("Three was what you typed.").
        action(N):- write("I don't know that number").
GOAL
        Write("Type a number from 1 to 3: "),
        readint(Num),
        action(Num), nl.
```

Çalıştırılmadığı sürece *Cut*'ın bir etkisi olmayacaktır. Yeni bir *Cut* gerçekleştirmek için Prolog *Cut*'ın bulunduğu kuralın içine girmeli ve *Cut*'ın bulunduğu yere ulaşmalıdır.

Programın bir önceki halinde tüm kurallar istenilen sıra ile yazılabilirken *Cut* kullanılan durumda "I don't know that number." yazısının yazdırıldığı kuralın diğer tüm kurallar denenmeden denenmeyeceğine emin olunmalıdır. Bu programda kullanılan *Cut*'lar programın mantığını değiştiren kırmızı *Cut*'lardır. Eğer X<>1, X<>2 ve X<>3 testlerini çıkarmadan her clause'a *Cut* eklenirse sadece zaman kazanmaya yarayan yeşil *Cut* kullanılmış olur. Kazanılan verimlilik fazla değildir ama hata yapma riski daha azdır.

Cut güçlü ama sorun yaratabilen bir Prolog işlemidir. Diğer programlama dillerindeki GoTo işlemini andırdığı için çok fazla kullanılabileceği durum vardır ama programları anlaşılması güç hale getirmesi de mümkündür.

4.5. Hesaplanan Değerleri Geri Döndürme

Bir Prolog kuralı (rule) ya da gerçeği (fact) kendisini çağıran goal'e bilgi döndürebilir.bu işlem daha önceden bağlanmamış argümanları değişkenlerle bağlamak sureti ile olur.

likes(bill,cindy) gerçeği

likes(bill,Who) goal'üne Who değişkeni ile cindy bilgisini bağlamak sureti ile bilgi döndürür. Bir kuralda da bilgi döndürme işlemi aynı şekilde gerçekleşir.

```
PREDICATES
        nondeterm classify(integer,symbol)
CLAUSES
        classify(0,zero).
        classify(X,negative):- X<0.
        classify(X,positive):- X>0.
```

Classify kuralının ilk argümanı her zaman bir sabit ya da bağlı (bounded) değişken olmalıdır. İkinci argümanı ise bağlı veya bağlanmamış olabilir; ilk argümanın değerine göre "zero", "negative" ya da "positive" symbol'ü ile match edilecektir.

45 sayısının positive olup olmadığı şöyle bir goal ile sorgulanabilir:

```
Goal        classify(45,positive).
yes
```

45 sayısı 0'dan büyük olduğundan üçüncü classify clause'u başarılı olacaktır. İkinci argüman 45 sayısı için positive bilgisi ile match edilmelidir ve sorgulanan goal'de durum zaten bu şekilde verilmiştir. Dolayısıyla eşleşme başarılı olur ve cevap "yes" olarak verilir.eğer goal

Goal classify(45,negative).

olarak verilirse Prolog ilk clause'u deneyecek ve 0 ya da ikinci argüman olan zero ile uyuşma sağlayamayacaktır. İkinci clause denendiğinde X<0 testi 45 sayısı için başarısız olur. Son clause da denendiğinde bu sefer ikinci argümanlar uyuşmayacaktır.

Yes ya da No dışında bir cevap alabilmek için classify, ikinci argümanı boş (bağlanmamış) şekilde çağrılmalıdır.

Goal classify(45,What).

Bu goal'den dönecek olan sonuç şu şekilde olacaktır:

What=positive

1 Solution

Bu sonucu elde etmek için program şu aşamalardan geçer:

Goal ilk clause'un argümanlarıyla match etmez (uyuşmaz) çünkü ilk clause 0 durumu için verilmiştir.

İkinci clause'da 45, boşta olan X değişkeni ile, What ise negative değeri ile bağlanır. X<0 testi başarısız olunca (45<0 yanlıştır.) bu clause'dan çıkılır, yapılan bağlamalar bozulur.

Üçüncü clause'da yine 45 X ile, What ise positive verisi ile bağlanır. Bu sefer X>0 testi başarılı olur. bu başarılı bir çözüm olduğundan Prolog backtracking yapmaz, kendisini çağıran prosedüre What değişkenine bağladığı ve başarılı olan positive sonucu ile döner. X değişkeni de zaten verilen goal'e ait olduğundan goal yapılan değişken-veri bağlarını kullanabilir ve bulunan değer sonuç olarak ekrana basılır.

Kaynakça:

-Computational Logic Course Material (http://clip.dia.fi.upm.es/~logalg/)
- A. Aaby (http://burks.brighton.ac.uk/burks/pcinfo/progdocs/plbook/) Logic Programming Chapter
- LOGIC, PROGRAMMING AND PROLOG (2ED) Ulf Nilsson and Jan Ma luszy_nski
- Peter D. Mosses (http://www.brics.dk/~pd)
- Prolog Programming slides written by Dr W.F. Clocksin