

# CENG 218 Programlama Dilleri

## Bölüm 4: Mantıksal Programlama

Öğr.Gör. Şevket Umut Çakır

Pamukkale Üniversitesi

Hafta 6

# Hedefler

- Mantıksal programlamanın doğasını anlamak
- Horn cümlelerini anlamak
- Çözümlemeyi ve birleştirmeyi anlamak
- Prolog diline aşina olmak
- Mantıksal programlamayla ilgili sorunları keşfetmek
- Curry diline aşina olmak



# Giriş

- **Mantık(Logic):** akıl yürütme ve ispat bilimi
  - ▶ Antik Yunan filozoflarının zamanından beri var
- Matematiksel veya sembolik mantık: 1800'lerin ortalarında George Boole ve Augustus De Morgan ile başladı
- Mantık, bilgisayarlar ve programlama dilleriyle yakından ilişkilidir
  - ▶ Devreler Boole cebri kullanılarak tasarlanmıştır
  - ▶ Mantıksal ifadeler, programlama dillerinin semantiği olan **aksiyomatik semantiği(axiomatic semantics)** tanımlamak için kullanılır



# Giriş

- Mantıksal ifadeler kurallı belirtim(formal specification) olarak kullanılabilir
- Aksiyomatik semantikle birlikte, bir programın doğruluğunu tamamen matematiksel bir şekilde kanıtlamak için kullanılabilirler
- Bilgisayarlar matematiksel mantık ilkelerini uygulamak için kullanılır
  - ▶ **Otomatik indirgeme sistemleri(Automatic deduction systems)** veya **otomatik teorem kanıtlayıcıları(automatic theorem provers)**, ispatları hesaplamaya dönüştürür
  - ▶ Hesaplama bir tür kanıt olarak görülebilir
  - ▶ **Prolog** programlama diline yol açtı



# Mantık ve Mantıksal Programlar

- Matematiksel mantığı anlamalı
- **Birinci dereceden yüklem hesabı(First-order predicate calculus):** mantıksal ifadeleri resmi olarak ifade etmenin bir yolu
- **Mantıksal ifadeler(Logical statements):** doğru veya yanlış olan ifadeler
- **Aksiyomlar(Axioms):** diğer doğru ifadelerin kanıtlanabileceği durumda doğru olduğu varsayılan mantıksal ifadeler



# Mantık ve Mantıksal Programlar

- Birinci dereceden yüklem hesap ifadesi bölümleri:
  - ▶ **Sabitler(Constants)**: genellikle sayılar veya isimler
  - ▶ **Yüklemler(Predicates)**: doğru veya yanlış fonksiyonların adları
  - ▶ Fonksiyonlar: Boole olmayan değerler döndüren fonksiyonlar
  - ▶ **Henüz belirtilmemiş miktarları temsil eden değişkenler(Variables that stand for as yet unspecified quantities )**
  - ▶ **Bağlayıcılar(Connectives)**: ve veya değil, çıkarım(implication) ( $\rightarrow$ ) denkliliği(equivalence) ( $\leftrightarrow$ ) gibi işlemler
  - ▶ **Nicelik belirteçleri(Quantifiers)**: değişkenleri tanıtan işlemler
  - ▶ **Noktalama sembolleri(Punctuation symbols)**: parantez, virgül, nokta



# Mantık ve Mantıksal Programlar

- Örnek 1:

- ▶ Aşağıdaki ifadeler mantıksal ifadelerdir

- 0 bir doğal sayıdır.

- 2 bir doğal sayıdır.

- Bütün  $x$ 'ler için, eğer  $x$  bir doğal sayıysa,  $x$ 'i takip eden(successor) sayı da doğaldır.

- 1 bir doğal sayıdır.

- ▶ Yükleme hesabına çeviri:

- `natural(0).`

- `natural(2).`

- `For all  $x$ ,  $\text{natural}(x) \rightarrow \text{natural}(\text{successor}(x)).$`

- `natural(-1)`



# Mantık ve Mantıksal Programlar

- İlk ve üçüncü ifadeler aksiyomlardır
- İkinci ifade kanıtlanabilir
$$2 = \text{successor}(\text{successor}(0)) \text{ and } \text{natural}(0) \rightarrow$$
$$\hookrightarrow \text{natural}(\text{successor}(0)) \rightarrow$$
$$\hookrightarrow \text{natural}(\text{successor}(\text{successor}(0)))$$
- Dördüncü cümle aksiyomlardan kanıtlanamaz, bu nedenle yanlış olduğu varsayılabilir
- Üçüncü ifadedeki  $x$ , henüz belirlenmemiş bir miktarı temsil eden bir değişkendir





# Mantık ve Mantıksal Programlar

- **Evrensel niceleyici(Universal quantifier)**: yüklemeler arasındaki ilişki, değişken tarafından adlandırılan evrendeki her şey için doğrudur
  - ▶ Örn: tüm  $x$ 'ler için(for all  $x$ )
- **Varoluşsal niceleyici(Existensial quantifier)**: bir yüklem, evrendeki değişkenle gösterilen en az bir şey için doğrudur
  - ▶ bir  $x$  vardır(there exists  $x$ )
- Nicelik belirteci tarafından eklenen bir değişkenin nicelik belirteci tarafından **bağlandığı(bound)** söylenir



# Mantık ve Mantıksal Programlar

- Nicelik belirteci ile bağlı olmayan bir değişkenin **serbest(free)** olduğu söylenir
- Yüklemlere ve fonksiyonlara yönelik bağımsız değişkenler yalnızca **terimler(terms)** olabilir: değişkenlerin, sabitlerin ve fonksiyonların kombinasyonları
  - ▶ Terimler, yüklemler, nicelik belirleyiciler veya bağlantılar içeremez



# Mantık ve Mantıksal Programlar

- Örnek 2:

- ▶ At bir memelidir.
- ▶ İnsan bir memelidir.
- ▶ Memelilerin dört bacağı vardır ve kolları yoktur, ya da iki bacağı ve iki kolu vardır.
- ▶ Atın kolu yoktur.
- ▶ İnsanın kolları vardır.
- ▶ İnsanın bacakları yoktur.

- Yükleme hesabına çeviri:

mammal(horse).

mammal(human).

for all x, mammal(x)  $\rightarrow$  legs(x, 4) and arms(x, 0) or  
 $\hookrightarrow$  legs(x, 2) and arms(x, 2).

arms(horse, 0).

not arms(human, 0).

legs(human, 0).



# Mantık ve Mantıksal Programlar

- Birinci dereceden yüklem hesabı(First-order predicate calculus) da çıkarım kurallarına sahiptir
- **Çıkarım kuralları(Inference rules)**: belirli bir ifade kümesinden yeni ifadeler türetmenin veya kanıtlamanın yolları
- Örnek:  $a \rightarrow b$  ve  $b \rightarrow c$  ifadelerinden, resmi olarak şu şekilde yazılan  $a \rightarrow c$  ifadesi türetilir:

$$\frac{a \rightarrow b \text{ ve } b \rightarrow c}{a \rightarrow c}$$



# Mantık ve Mantıksal Programlar

- Örnek 2'den şu ifadeleri türetebiliriz:  
legs(horse, 4).  
legs(human, 2).  
arms(human, 2).
- **Teoremler(Theorems)**: aksiyoimlardan türetilen ifadeler
- **Mantıksal programlama(Logic programming)**: bir dizi ifadenin aksiyoimler olduğu varsayılır ve bunlardan, çıkarım kurallarının otomatik bir şekilde uygulanmasıyla istenen bir gerçek elde edilir



# Mantık ve Mantıksal Programlama

- **Mantıksal programlama dili (Logic programming language):** çıkarım kurallarını uygulamak için belirli algoritmalarla birlikte mantıksal ifadeler yazmak için bir notasyon sistemi
- **Mantıksal program (Logic program):** aksiyom olarak kabul edilen mantıksal ifadeler kümesi
- Türetilecek ifadeler, hesaplamayı başlatan girdi olarak görülebilir.
  - ▶ **Sorgular (Queries)** veya **hedefler (goals)** olarak da adlandırılır



# Mantık ve Mantıksal Programlar

- Mantıksal programlama sistemleri bazen **tümdengelimli veritabanları(deductive databases)** olarak adlandırılır
  - ▶ Bir dizi ifadeden ve sorgulara yanıt verebilecek bir indirgeme(deduction) sisteminden oluşur
  - ▶ Sistem, çıkarımları içeren gerçekler(facts) ve sorgular(queries) hakkındaki soruları yanıtlayabilir
- **Kontrol problemi(Control problem)**: bir ifade türetmek için kullanılan belirli bir yol veya adım dizisi



# Mantık ve Mantıksal Programlar

- Mantıksal programlama paradigması (Kowalski)[1]:
  - ▶ algoritma = mantık + kontrol
- Zorunlu programlama (Wirth)[2] ile karşılaştırılırsa:
  - ▶ algoritmalar = veri yapıları + programlar
- Mantıksal programlar kontrolü ifade etmediğinden, teoride işlemler herhangi bir sırada veya eşzamanlı olarak gerçekleştirilebilir
  - ▶ Mantıksal programlama dilleri paralellik için doğal adaylardır





# Mantık ve Mantıksal Programlar

- Mantıksal programlama sistemlerinde sorunlar var
- Otomatik indirgeme sistemleri, tüm birinci dereceden yüklem hesabını işlemekte güçlük çekiyor
  - ▶ Aynı ifadeleri ifade etmenin çok fazla yolu
  - ▶ Çok fazla çıkarım kuralı
- Mantıksal programlama sistemlerinin çoğu, kendilerini Horn cümleleri(Horn clauses) adı verilen belirli bir yüklem hesabı alt kümesiyle sınırlar



# Horn Cümleleri

- **Horn cümlesi(clause):**  $a_1$  ve  $a_2$  ve  $a_3 \dots$  ve  $a_n \rightarrow b$  biçimindeki bir ifade
- $a_i$ 'lerin yalnızca basit ifadeler olmasına izin verilir
  - ▶ veya bağlayıcılarına ve *niceleyicilere* izin verilmez
- Bu ifade  $b$ 'nin  $a_1$  ile  $a_n$  arasındaki ifadelerden çıkarım yapıldığını(imply), veya bütün  $a_i$ 'ler doğru ise  $b$ 'nin doğru olduğunu belirtir
  - ▶  $b$  cümlenin **başıdır(head)**
  - ▶  $a_1, a_2, \dots, a_n$  cümlenin **gövdesidir(body)**
- Eğer hiç  $a_i$  yoksa cümle  $\rightarrow b$  şekline dönüşür
  - ▶  $b$  her zaman doğrudur ve **gerçek(fact)** olarak adlandırılır



# Horn Cümleleri

- Horn cümleleri, mantıksal ifadelerin hepsini olmasa da çoğunu ifade etmek için kullanılabilir
- Örnek 4: birinci dereceden yüklem hesabı:

`natural(0).`

`for all x, natural(x)  $\rightarrow$  natural(successor(x)).`

- Nicelik belirtecini kaldırarak bunları Horn cümlelerine çevirebiliriz:

`natural(0).`

`natural(x)  $\rightarrow$  natural(successor(x)).`



# Horn Cümleleri

- Örnek 5: İki pozitif tamsayı  $u$  ve  $v$ 'nin en büyük ortak böleni için Öklid algoritmasının mantıksal açıklaması:  
 $u$  ve  $0$ 'ın OBEB'i  $u$ 'dur.  
 $u$  ve  $v$ 'nin OBEB'i,  $v$  sıfırdan farklı ise,  $v$  ile  $u$ 'nun  
 $\rightarrow v$ 'ye bölümünden kalan sayının OBEB'i ile aynıdır.
- Birinci dereceden yüklem hesabı:  
 for all  $u$ ,  $\text{gcd}(u, 0, u)$ .  
 for all  $u$ , for all  $v$ , for all  $w$ , not zero( $v$ ) and  $\text{gcd}(v$ ,  
 $\rightarrow u \bmod v, w) \rightarrow \text{gcd}(u, v, w)$ .



# Horn Cümleleri

- $\text{gcd}(u, v, w)$  'nin,  $w$ 'nun  $u$  ve  $v$ 'nin OBEB'i olduğunu ifade eden bir yüklem olduğuna dikkat edin
- Nicelik belirteçlerini kaldırarak Horn cümlelerine çevirirsek:  
 $\text{gcd}(u, 0, u).$   
 $\text{not zero}(v) \text{ and } \text{gcd}(v, u \bmod v, w) \rightarrow \text{gcd}(u, v, w).$



# Horn Cümleleri

- Örnek 6: mantıksal ifadeler

$x, y$ 'nin ebeveyni olan birinin ebeveyni ise,  $y$ 'nin  
 $\hookrightarrow$  büyük ebeveynidir( $\text{grandparent}$ ).

- Yükleme hesabı:

for all  $x$ , for all  $y$ , (there exists  $z$ ,  $\text{parent}(x, z)$  and  
 $\hookrightarrow \text{parent}(z, y) \rightarrow \text{grandparent}(x, z)$ ).

- Horn cümlesi:

$\text{parent}(x, z) \text{ and } \text{parent}(z, y) \rightarrow \text{grandparent}(x, z)$ .



# Horn Cümleleri

- Örnek 7: Mantıksal ifadeler

Tüm  $x$ 'ler için, eğer  $x$  bir memeliyse,  $x$ 'in iki veya  
 $\rightarrow$  dört bacağı vardır

- Yüklemler hesabı:

for all  $x$ ,  $\text{mammal}(x) \rightarrow \text{legs}(x, 2) \text{ or } \text{legs}(x, 4)$ .

- Bu, aşağıdaki Horn cümleleri ile yaklaştırılabilir:

$\text{mammal}(x) \text{ and } \text{not } \text{legs}(x, 2) \rightarrow \text{legs}(x, 4)$ .

$\text{mammal}(x) \text{ and } \text{not } \text{legs}(x, 4) \rightarrow \text{legs}(x, 2)$ .

- Genel olarak, bir  $\rightarrow$  bağlantısının sağında ne kadar çok bağlayıcı görünürse, bir Horn cümleleri kümesine çevirmek o kadar zor olur



# Horn Cümleleri

- **Prosedürel yorumlama(Procedural interpretation):** Horn cümleleri bir prosedür olarak görmek için tersine çevrilebilir

$b \leftarrow a_1$  ve  $a_2$  ve  $a_3 \dots$  ve  $a_n$

- Bu, b prosedürü haline gelir, burada gövde,  $a_i$ 'lerin gösterdiği işlemlerdir.
  - ▶ Bağılamdan bağımsız gramer(context-free grammar) kurallarının özyinelemeli iniş ayrıştırırmada(recursive descent parsing) prosedür tanımları olarak yorumlanma şekline benzer
  - ▶ Mantıksal programlar doğrudan ayrıştırıcıları(parser) oluşturmak için kullanılabilir





# Horn Cümleleri

- Doğal dilin ayrıştırılması, Prolog'un orijinal gelişimi için bir motivasyondur
- **Belirli cümle gramerleri(Definite clause grammars):** Prolog programlarında kullanılan belirli türdeki gramer kuralları
- Horn cümleleri, katı bir şekilde uygulamalardan ziyade prosedürlerin **özellikleri(specifications)** olarak da görülebilir
- Örnek: bir sıralama prosedürünün özelliği:  
 $\text{sort}(x, y) \leftarrow \text{permutation}(x, y) \text{ and sorted}(y).$



# Horn Cümleleri

- Horn cümleleri algoritmaları sağlamaz, sadece sonucun sahip olması gereken özellikleri sağlar
- Çoğu mantıksal programlama sistemi, Horn cümlelerini geriye doğru yazar ve ve bağlantılarını bırakır:

$\text{gcd}(u, 0, u).$

$\text{gcd}(u, v, w) \leftarrow \text{not zero}(v), \text{gcd}(v, u \bmod v, w).$

- **gcd** için standart programlama dili ifadesine benzerliğe dikkat edin:

$\text{gcd}(u, v) = \text{if } v = 0 \text{ then } u \text{ else } \text{gcd}()v, u \bmod v$



# Horn Cümleleri

- Değişken kapsamı:
  - ▶ Başta kullanılan değişkenler parametre olarak görüntülenebilir
  - ▶ Yalnızca gövdede kullanılan değişkenler yerel, geçici değişkenler olarak görülebilir
- Sorgular veya hedef ifadeleri: bir gerçeğin tam tersi
  - ▶ Başsız Horn cümleleri
  - ▶ Örnekler:  
`mammal(human) ← . --- bir gerçek`  
`mammal(human) . --- bir sorgu veya hedef`



# Çözümleme ve Birleştirme

- **Çözümleme(Resolution):** Horn cümleleri için bir çıkarım kuralı
  - ▶ Birinci Horn cümlesinin başı, ikinci Horn cümlesinin gövdesindeki ifadelerden biriyle eşleşirse, ikincinin gövdesindeki ifade birincinin gövdesi ile değiştirebilir
- Örnek: verilen iki Horn cümlesi:
 
$$a \leftarrow a_1, \dots, a_n$$

$$b \leftarrow b_1, \dots, b_n$$
  - ▶  $b_i$ ,  $a$  ile eşleşirse, bu cümle sonucunu çıkarabiliriz:
 
$$b \leftarrow b_1, \dots, b_{i-1}, a_1, \dots, a_n, b_{i+1}, \dots, b_n$$



# Çözümleme ve Birleştirme

- Örnek: verilen  $b \leftarrow a$  ve  $c \leftarrow b$ 
  - ▶ Çözümleme  $c \leftarrow a$  der
- Başka bir yol: her iki cümlenin sol ve sağ taraflarını birleştirin ve her iki tarafla eşleşen ifadeleri iptal edin
- Örnek: verilen  $b \leftarrow a$  ve  $c \leftarrow b$ 
  - ▶ Birleştir:  $b, c \leftarrow a, b$
  - ▶ Her iki taraftaki  $b$ 'leri iptal et:  $c \leftarrow a$



# Çözümleme ve Birleştirme

- Mantık işleme sistemi, bir hedefi(goal) eşleştirmek ve onu gövdeyle değiştirmek için bu süreci kullanır ve **alt hedefler(subgoals)** adı verilen yeni bir hedef listesi oluşturur
- Tüm hedefler sonunda ortadan kaldırılırsa, boş Horn cümlesi türetilirse, orijinal ifade kanıtlanmıştır
- İfadeleri değişkenlerle eşleştirmek için değişkenleri terimlere eşitleyerek ifadeleri aynı yapın ve ardından her iki taraftan iptal edin
  - ▶ Bu sürece **birleştirme(unification)** denir
  - ▶ Bu şekilde kullanılan değişkenlerin **somutlaştırıldığı(instantiated)** söylenir



# Çözümleme ve Birleştirme

- Örnek 10: Çözümleme ve birleştirme ile  $\text{gcd}$   
 $\text{gcd}(u, 0, u)$ .  
 $\text{gcd}(u, v, w) \leftarrow \text{not zero}(v), \text{gcd}(v, u \bmod v, w)$ .
- Hedef:  
 $\leftarrow \text{gcd}(10, 15, x)$ .
- Çözümleme birinci cümleyle başarısız olur (10, 0 ile eşleşmez), bu nedenle ikinci cümleyi kullanır ve birleştirir:  
 $\text{gcd}(10, 15, x) \leftarrow \text{not zero}(10), \text{gcd}(10, 15 \bmod 10, x),$   
 $\hookrightarrow \text{gcd}(15, 10, x)$ .



# Çözümleme ve Birleştirme

## • Örnek 10(devam):

- ▶ Eğer `zero(10)` yanlışsa `not zero(10)` doğrudur
- ▶  $15 \bmod 10$ 'u 5'e sadeleştirin ve iki taraftaki `gcd(15, 10, x)` ifadelerini iptal edin:  
 $\leftarrow \text{gcd}(10, 5, x).$
- ▶ Üstteki gibi birleştirilirse:  
 $\text{gcd}(10, 5, x) \leftarrow \text{not zero}(5), \text{gcd}(5, 10 \bmod 5, x),$   
 $\hookrightarrow \text{gcd}(10, 5, x).$
- ▶ Bu alt hedefi elde etmek için:  
 $\leftarrow \text{gcd}(5, 0, x).$
- ▶ Bu artık ilk kuralla eşleşir, bu nedenle  $x$ 'i 5 olarak almak boş ifadeyi verir





# Çözümleme ve Birleştirme

- Mantıksal programlama sistemi, şunları belirten sabit bir algoritmaya sahip olmalıdır:
  - ▶ Bir hedefler listesini çözme girişiminde bulunma sırası
  - ▶ Hedefleri çözmek için cümlelerin kullanım sırası
- Bazı durumlarda sıralama, bulunan cevaplar üzerinde önemli bir etkiye sahip olabilir
- Önceden tanımlanmış sıralamayla Horn cümleleri ve çözümleme kullanan mantıksal programlama sistemleri, programcının sistemin cevapları üretme şeklinin farkında olmasını gerektirir



# Prolog Dili

- **Prolog**: en yaygın kullanılan mantıksal programlama dili
  - ▶ Horn cümlelerini kullanır
  - ▶ Çözümleme işlemini katı bir derinine arama(depth-first search) stratejisi ile sağlar
- Artık Prolog için bir ISO standardı var
  - ▶ 1970'lerin sonu ve 1980'lerin başında geliştirilen Edinburgh Prolog versiyonuna dayanmaktadır



# Gösterim ve Veri Yapıları

- Prolog gösterimi, Horn cümleleri ile neredeyse aynıdır
  - ▶ Çıkarım oku  $\leftarrow$ ,  $:-$  olur
  - ▶ Değişkenler büyük harf, sabitler ve isimler küçük harftir
  - ▶ Çoğu uygulamada, önde gelen alt çizgi olan bir değişkeni de gösterebilir
  - ▶ **ve** için virgül kullanılır, **veya** için noktalı virgül kullanılır
  - ▶ Liste, virgülle ayrılmış öğelerle köşeli parantezlerle yazılmıştır
  - ▶ Listeler terimler veya değişkenler içerebilir



# Gösterim ve Veri Yapıları

- Dikey çubuk kullanarak listenin başı ve kuyruğu ayrılabilir
- Örnek:  $[H|T] = [1, 2, 3]$  ifadesi  $H=1$ ,  $T=[2, 3]$  anlamına gelir
- Örnek:  $[X, Y|Z] = [1, 2, 3]$  ifadesi,  $X=1$ ,  $Y=2$ , **and**  $Z=[3]$  anlamına gelir
- Yerleşik yüklemeler arasında **not**, **=** ve **read**, **write** ve **nl** (satırsonu) gibi G / Ç işlemleri bulunur
- Küçük eşittir operatörü, çıkarım ile karışıklığı önlemek için **=<** şeklinde kullanılır



# Prolog'da Yürütme

- Prolog sistemlerinin çoğu yorumlayıcıdır
- Prolog programı şunlardan oluşur:
  - ▶ Prolog sözdizimindeki Horn cümleleri kümesi, genellikle bir dosyadan girilir ve dinamik olarak tutulan bir cümle veritabanında saklanır
  - ▶ Bir dosyadan veya klavyeden girilen hedefler kümesi
- Çalışma zamanında, Prolog sistemi bir sorgu isteyecektir



# Prolog'da Yürütme

- Örnek 11: Veritabanına girilen cümleler

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```

- Sorgular:

```
?- ancestor(amy, bob).
true .
?- ancestor(bob, amy).
false.
?- ancestor(X, bob).
X = amy .
?- ancestor(X, bob).
X = amy ;
X = bob.
```

- ; veya anlamına gelir, satırbaşı(carriage return/enter) devam eden aramayı durdurur



# Aritmetik

- Prolog yerleşik aritmetik işlemlere sahiptir
  - ▶ Terimler infix veya prefix gösterimi ile yazılabilir
- Prolog, bir terimin aritmetik veya kesinlikle veri olduğunu söyleyemez
- Değerlendirmeyi zorlamak için, yerleşik `is` yüklemi kullanılmalıdır

```
?- write(3+5).
```

```
3+5
```

```
true.
```

```
?- X is 3+5, write(X).
```

```
8
```

```
X = 8.
```



# Aritmetik

- En büyük ortak bölen algoritması

- ▶ Genel Horn cümlelerinde:

`gcd(u, 0, u).`

`gcd(u, v, w) ← not zero(v), gcd(v, u mod v, w).`

- ▶ Prolog'da:

`gcd(U, 0, U).`

`gcd(U, V, W) :- not(V = 0), R is U mod V, gcd(V, R, W).`





# Birleştirme(Unification)

- **Birleştirme(Unification)**: değişkenlerin çözümleme sırasında eşleşecek şekilde somutlaştırıldığı süreç
  - ▶ Semantiği birleştirme ile belirlenen temel ifade eşitliktir
- Prolog'un birleştirme algoritması:
  - ▶ Sabit sadece kendisiyle birleşir
  - ▶ Gerçekleştirilmemiş(değer kazanmamış) değişken herhangi bir şeyle birleşir ve o şeye somutlaştırılır
  - ▶ Yapılandırılmış terim (argümanlara uygulanan fonksiyon), yalnızca aynı işlev adı ve aynı sayıda bağımsız değişken varsa başka bir terimle birleşir



# Birleştirme(Unification)

?- me = me.  
true.

?- me = you.  
false.

?- me = X.  
X = me.

?- f(a, X) = f(Y, b).  
X = b,  
Y = a.

?- f(X) = g(X).  
false.

?- f(X) = f(a, b).  
false.

?- f(a, g(X)) = f(Y, b).  
false.

?- f(a, g(X)) = f(Y, g(b)).  
X = b,  
Y = a.



# Birleştirme(Unification)

- Birleştirme, doğrulanmamış değişkenlerin belleği paylaşmasına (birbirlerinin takma adları haline gelmesine) neden olur
- Örnek: iki doğrulanmamış değişken birleştirilmiş

```
?- X=Y, writeln(X), writeln(Y).
```

```
_13582
```

```
_13582
```

```
X = Y.
```

- **Örüntü yönelimli çağrı(Pattern-directed invocation):** Değişken yerine bir kalıp kullanmak, onu bir hedefte o yerde kullanılan bir değişkenle birleştirir.

► Örnek:

```
cons(X, Y, [X|Y]).
```



# Birleştirme(Unification)

- **Append** prosedürü:

`append(X, Y, Z) :- X=[], Y = Z.`

`append(X, Y, Z) :- X = [A|B], Z = [A|W], append(B, Y, W).`

- Birinci cümle: boş listeye bir liste eklemek bu listeyi verir
- İkinci cümle: başı A ve kuyruğu B olan bir listeyi bir Y listesine eklemek, başı da A ve kuyruğu Y'nin B'ye eklenmiş hali olan ve bir liste verir



# Birleştirme(Unification)

- **append** prosedürü daha kısaca yeniden yazılmıştır:

```
append([], Y, Y).
```

```
append([A|B], Y, [A|W]) :- append(B, Y, W).
```

- Ekleme ayrıca geriye doğru çalıştırılabilir ve belirli bir listeyi almak için iki liste eklemenin tüm yollarını bulabilir:

```
?- append(X, Y, [1,2]).
```

```
X = [],
```

```
Y = [1, 2] ;
```

```
X = [1],
```

```
Y = [2] ;
```

```
X = [1, 2],
```

```
Y = [] ;
```

```
false.
```



# Birleştirme(Unification)

- **reverse** prosedürü:

```
reverse([], []).
```

```
reverse([H|T], L) :- reverse(T, L1), append(L1, [H], L).
```



# Birleştirme(Unification)

```
gcd(U, 0, U).
```

```
gcd(U, V, W) :- not(V=0), R is U mod V, gcd(V, R, W).
```

```
append([], Y, Y).
```

```
append([A|B], Y, [A|W]) :- append(B,Y,W).
```

```
reverse([], []).
```

```
reverse([H|T], L) :- reverse(T,L1), append(L1, [H], L).
```

Şekil: gcd, append ve reverse için Prolog cümleleri



# Prolog'un Arama Stratejisi

- Prolog, çözümlemeyi kesinlikle doğrusal bir şekilde uygular
  - ▶ Hedefleri soldan sağa değiştirir
  - ▶ Veritabanındaki cümleleri yukarıdan aşağıya doğru ele alır
  - ▶ Alt hedefler hemen değerlendirilir
  - ▶ Bu arama stratejisi, olası seçenekler ağacında derinlemesine bir arama(depth-first search) ile sonuçlanır

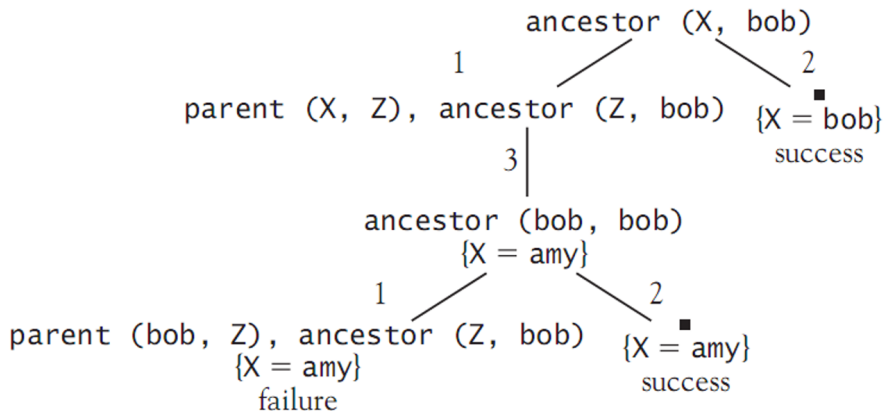
- Örnek:

```
1 ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).  
2 ancestor(X, X).  
3 parent(amy, bob).
```





# Prolog'un Arama Stratejisi



**Figure 4.2** A Prolog search tree showing subgoals, clauses used for resolution, and variable instantiations

# Prolog'un Arama Stratejisi

- Ağaçtaki yaprak düğümleri, en soldaki cümle için bir eşleşme bulunmadığında veya tüm cümlecikler ortadan kaldırıldığında (başarılı) ortaya çıkar
- Başarısız olursa veya kullanıcı noktalı virgülle devam eden bir arama gösterirse, Prolog daha fazla yol bulmak için ağacı **geriye doğru izler(backtracks)**
- Derinine arama stratejisi etkilidir: yığın tabanlı veya yinelemeli bir şekilde uygulanabilir
  - ▶ Arama ağacının sonsuz derinlikte dalları varsa sorunlu olabilir



# Prolog'un Arama Stratejisi

- Örnek: farklı sıradaki aynı cümlecikler
  - 1 `ancestor(X, Y) :- ancestor(Z, Y), parent(X, Z).`
  - 2 `ancestor(X, X).`
  - 3 `parent(amy, bob).`
- Prolog'un `ancestor(Z, Y)` sorgusunu tatmin etmeye çalışırken sonsuz bir döngüye girmesine neden olur ve sürekli olarak ilk cümleyi tekrar kullanır
- Enine arama(Breadth-first search), varsa her zaman çözüm bulur
  - ▶ Derinine aramadan çok daha maliyetli, bu yüzden kullanılmıyor



# Döngüler ve Kontrol Yapıları

- Döngüler ve tekrarlayan aramalar gerçekleştirmek için Prolog'un geri izlemesi(backtracking) kullanabilir
  - ▶ Yerleşik **fail** koşulu kullanılarak bir çözüm bulunduğunda bile geri izlemeyi zorlamalıdır

- Örnek:

```
printpieces(L) :- append(X, Y, L), write(X), write(Y),
    ↪ nl, fail.
```

```
?- printpieces([1, 2]).
```

```
[] [1,2]
```

```
[1] [2]
```

```
[1,2] []
```

```
false.
```



# Döngüler ve Kontrol Yapıları

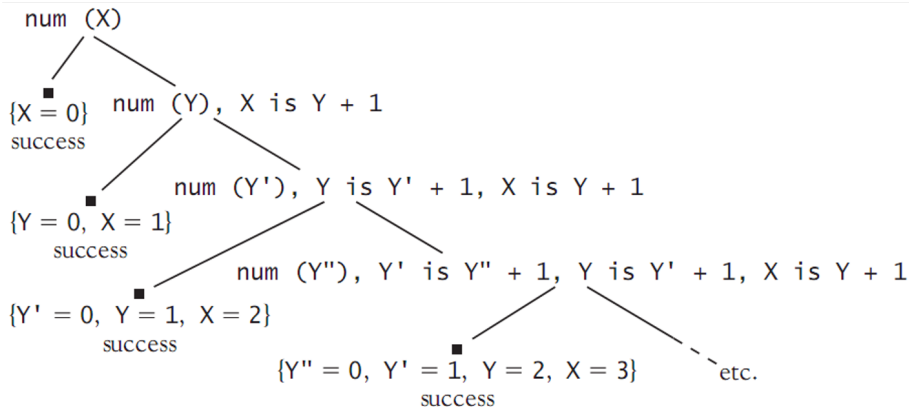
- Tekrarlayan hesaplamalar elde etmek için de bu teknik kullanılır
- Örnek: Bu maddeler, `num(X)` hedefine çözüm olarak 0'dan büyük veya 0'a eşit tüm tam sayıları üretir

```
1 num(0).  
2 num(X) :- num(Y), X is Y + 1.
```

- Arama ağacının sağında sonsuz bir dalı vardır



# Döngüler ve Kontrol Yapıları



**Figure 4.3** An infinite Prolog search tree showing repetitive computations



# Döngüler ve Kontrol Yapıları

- Örnek: 1'den 10'a kadar tamsayılar oluşturmaya çalışmak  
`writenum(I, J) :- num(X), I =< X, X =< J, write(X), nl,`  
`↪ fail.`
- `X =< 10` asla başarılı olmayacak olsa bile, `X = 10`'dan sonra sonsuz bir döngüye neden olur
- **kesme(cut)** operatörü (! olarak yazılır) karşılaşıldığında bir seçimi dondurur



# Döngüler ve Kontrol Yapıları

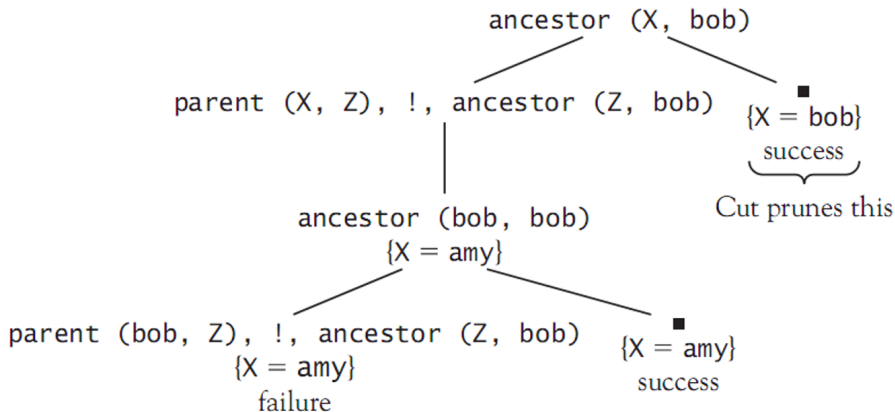
- Geri izlemede(backtracking) bir kesime(cut) ulaşırsa, ana düğümün alt ağaçlarının aranması durur ve arama büyükbaba(granparent) düğümle devam eder
  - ▶ Kesme operatörü, kesme işaretinin sağında yer alan bütün kardeş düğümleri(sibling) budar(prune)
- Örnek:
 

```
ancestor(X, Y) :- parent(X, Z), !, ancestor(Z, Y).
ancestor(X, X).
parent(amy, bob).
```
- $X = \text{bob}$  içeren dal budanacağından sadece  $X = \text{amy}$  bulunacaktır





# Döngüler ve Kontrol Yapıları



**Figure 4.4** Consequences of the cut for the search tree of Figure 4.2

# Döngüler ve Kontrol Yapıları

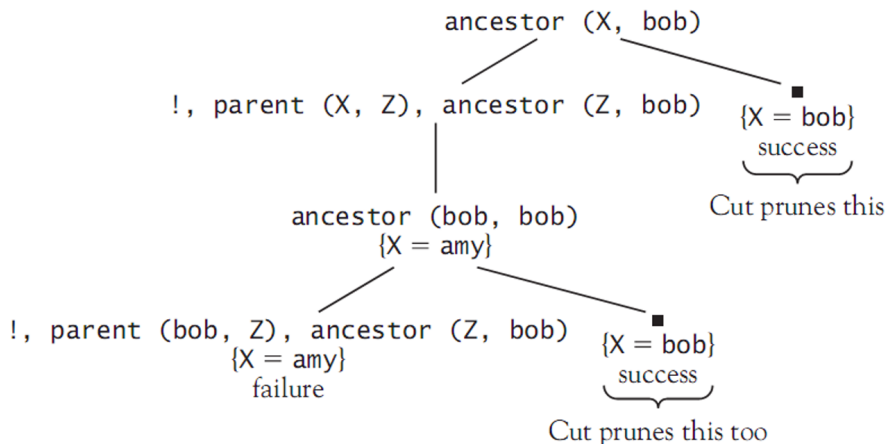
- Bu örneği tekrar yazarsak

```
ancestor(X, Y) :- !, parent(X, Z), ancestor(Z, Y).  
ancestor(X, X).  
parent(amy, bob).
```

- Hiç sonuç bulunmaz



# Döngüler ve Kontrol Yapıları



**Figure 4.5** A further use of the cut prunes all solutions from Figure 4.2

# Döngüler ve Kontrol Yapıları

- Yeniden yazılırsa:

```
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).
```

```
ancestor(X, X) :- !.
```

```
parent(amy, bob).
```

- ▶ `ancestor(X, bob)` budanmamış olduğundan her iki çözüm de bulunacaktır

- Kesme işareti, takip edilmesi gereken alt ağaçtaki dal sayısını azaltmak için kullanılabilir
- Ayrıca, daha önce gösterilen I ve J arasındaki sayıları yazdırmak için programdaki sonsuz döngü sorununu çözer



# Döngüler ve Kontrol Yapıları

- Daha önce gösterilen sonsuz döngüye bir çözüm:

```
num(0).
```

```
num(X) :- num(Y), X is Y + 1.
```

```
writenum(I, J) :- num(X), I =< X, X =< J, write(X), nl, X
```

- ▶  $X = J$ , üst sınır J'ye ulaşıldığında başarılı olur
- ▶ Kesme işlemi, geri izlemenin(backtracking) başarısız olmasına neden olacak ve X'in yeni değerlerinin aranmasını durduracaktır.



# Döngüler ve Kontrol Yapıları

- Ayrıca, emir kipi ve işlevsel dillerde if-else yapılarını taklit etmek için cut kullanabilir, örneğin:

$D = \text{if } A \text{ then } B \text{ else } C$

- Prolog kodu:

$D :- A, !, B.$

$D :- C.$

- Kesme olmadan neredeyse aynı sonucu elde edebilirdi, ancak A iki kez yürütülecekti

$D :- A, B.$

$D :- \text{not}(A), C.$



# Döngüler ve Kontrol Yapıları

```

primes(Limit, Ps) :- integers(2, Limit, Is), sieve(Is, Ps).
integers(Low, High, [Low|Rest]) :- Low =< High, !, M is Low+1,
    ↪ integers(M, High, Rest).
integers(Low, High, []).
sieve([], []).
sieve([I|Is], [I|Ps]) :- remove(I, Is, New), sieve(New, Ps).
remove(P, [], []).
remove(P, [I|Is], [I|Nis]) :- not(0 is I mod P), !, remove(P, Is,
    ↪ Nis).
remove(P, [I|Is], Nis) :- 0 is I mod P, !, remove(P, Is, Nis).

```



# Mantıksal Programlamayla İlgili Sorunlar

- Mantıksal programlamanın asıl amacı, programlamayı bir spesifikasyon etkinliği yapmaktır
  - ▶ Programcının yalnızca bir çözümün özelliklerini belirlemesine izin verin ve dil uygulamasının(implementation) çözümü hesaplamak için gerçek yöntemi sağlamasına izin verin
- **Bildirime dayalı programlama(Declarative programming):** program, sorunun nasıl çözüldüğünü değil, belirli bir soruna çözümün ne olduğunu açıklar
- Mantıksal programlama dilleri, özellikle Prolog, bu hedefi kısmen karşıladı





# Mantıksal Programlamayla İlgili Sorunlar

- Programcı, mantık programlama sistemleri tarafından kullanılan algoritmaların doğasındaki tuzakların farkında olmalıdır
- Programcı bazen, bir cut/fail döngüsü uygulamak için temeldeki geri izleme(backtracking) mekanizmasını kullanmak gibi, bir programın daha da düşük düzeyli bir perspektifini benimsemelidir



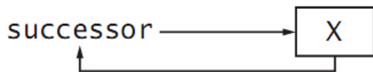
# Mantıksal Programlamayla İlgili Sorunlar

- Oluş kontrol problemi: Bir değişkeni bir terimle birleştirirken, Prolog değişkenin kendisinin somutlaştırıldığı terimde olup olmadığını kontrol etmez
- Örnek: `is_own_successor :- X = successor(X).`
- X'in kendi halefi(successor) olduğu bir X varsa bu doğru olacaktır
- Ancak, halef için başka herhangi bir cümle bulunmasa bile, Prolog, evet cevabını verir



# Mantıksal Programlamayla İlgili Sorunlar

- Bu, Prolog'a böyle bir X'i yazdırmaya çalışırsak ortaya çıkar:  
`is_own_successor :- X = successor(X).`
  - ▶ Prolog, sonsuz bir döngü ile yanıt verir, çünkü birleştirme X'i dairesel bir yapı olarak inşa etmiştir
  - ▶ Mantıksal olarak yanlış olması gereken şey şimdi bir programlama hatası olur



**Figure 4-7:** Circular structure created by unification



# Başarısızlık Olarak Olumsuzluk(Negation as Failure)

- **Kapalı dünya varsayımı(Closed-world assumption):** doğru olduğu kanıtlanamayan bir şeyin yanlış olduğu varsayılır
  - ▶ Tüm mantık programlama sistemlerinin temel bir özelliğidir
- **Başarısızlık olarak olumsuzluk(negation as failure):**  $X$  hedefi başarısız olduğunda  $\text{not}(X)$  hedefi başarılı olur
- Örnek: tek cümleli program: `parent(amy, bob)`.
- Eğer `?- not(mother(amy, bob))` sorgusu çalıştırılırsa
  - ▶ Cevap `true` olur, çünkü sistemin `mother` hakkında bir bilgisi yok
  - ▶ Eğer `mother` hakkında gerçekleri eklersek, bu artık doğru olmaz



# Başarısızlık Olarak Olumsuzluk

- **Monotonik olmayan akıl yürütme(Nonmonotonic reasoning)**: Bir sisteme bilgi eklemenin kanıtlanabilecek şeylerin sayısını azaltabileceği özellik
  - ▶ Bu, kapalı dünya varsayımının bir sonucudur
- Bununla ilgili bir sorun, başarısızlığın değişkenlerin somutlaştırılmasının geriye doğru izleme ile serbest bırakılmasına neden olmasıdır
  - ▶ Bir değişken, başarısızlıktan sonra artık uygun bir değere sahip olmayabilir



# Başarısızlık Olarak Olmumsuzluk

- Örnek: `human(bob) .` gerçeğini varsayar

```
?- human(X) .
```

```
X = bob .
```

```
?- not(not(human(X))), write(X) .
```

```
_11578
```

```
true .
```

- `not(human(X))` başarısız olduğu için `not(not(human(X)))` başarılı olur, ancak `X`'in `bob`'a somutlaştırılması serbest bırakılır



# Başarısızlık Olarak Olumsuzluk

- Örnek:

```
?- X=0, not(X=1).
```

```
X = 0.
```

```
?- not(X=1), X=0.
```

```
false.
```

- İkinci hedef çifti başarısız olur çünkü  $X$ ,  $X = 1$ 'in başarılı olması için 1'e örneklenir/somutlaştırılır(instantiate) ve sonra `not (X = 1)` başarısız olur
- $X = 0$  hedefine asla ulaşılmaz



# Horn Cümleleri Tüm Mantığı İfade Etmez

- Her mantıksal ifade Horn cümlelerine dönüştürülemez
  - ▶ Nicelik belirteçli ifadeler sorunlu olabilir

- Örnek:

*$p(a)$  and (there exists  $x$ ,  $\text{not}(p(x))$ )*

- Prolog kullanmaya çalışırken şunu yazabiliriz:

`p(a).`

`not(p(b)).`

- Bir hataya neden olur: `not` işlecini yeniden tanımlamaya çalışmak





# Horn Cümleleri Tüm Mantığı İfade Etmez

- Daha iyi bir yaklaşım basitçe  $p(a)$  olacaktır
  - ▶ Kapalı dünya yaklaşımı  $\text{not}(p(X))$ 'in  $a$ 'ya eşit olmayan tüm  $X$ 'ler için doğru olmasını zorlayacaktır
  - ▶ Ancak bu gerçekte,  $p(a)$  and  $(\text{for all } x, \text{not}(x = a) \rightarrow \text{not}(p(a)))$ . ifadesinin eşdeğeri olacaktır
  - ▶ Bu orijinal ifadeyle aynı değil



# Mantıksal Programlamada Kontrol Bilgileri

- Derinlemesine arama stratejisi ve hedeflerin ve ifadelerin doğrusal olarak işlenmesi nedeniyle, Prolog programları, programların başarısız olmasına neden olabilecek kontrol hakkında örtük bilgiler de içerir
  - ▶ Bir cümlelin sağ tarafının sırasını değiştirmek sonsuz döngüye neden olabilir
  - ▶ Cümlelerin sırasını değiştirmek tüm çözümleri bulabilir, ancak yine de daha fazla (var olmayan) çözüm arayarak sonsuz bir döngüye girebilir



# Mantıksal Programlamada Kontrol Bilgileri

```
sorted([]).  
sorted([X]).  
sorted([X, Y|Z]) :- X =< Y, sorted([Y|Z]).
```

```
permutation([], []).  
permutation(X, [Y|Z]) :- append(U, [Y|V], X), append(U, V,  
    ↪ W), permutation(W, Z).
```



# Mantıksal Programlamada Kontrol Bilgileri

- Bu, artan sırayla sıralanacak bir sayılar listesinin ne anlama geldiğinin matematiksel bir tanımıdır
  - ▶ Bir program olarak, olası en yavaş sıralamalardan biridir
  - ▶ Sıralanmamış listenin permütasyonları, biri sıralanana kadar üretilir
- Bir mantıksal programlama sisteminin matematiksel bir tanımı kabul etmesini ve onu hesaplamak için verimli bir algoritma bulmasını ister
- Bunun yerine, makul bir verimli sıralama elde etmek için algoritmadaki gerçek adımları belirtmeliyiz



# Mantıksal Programlamada Kontrol Bilgileri

```
qsort([], []).
```

```
qsort([H|T], S) :- partition(H, T, L, R), qsort(L, L1),  
    ↪ qsort(R, R1), append(L1, [H|R1], S).
```

```
partition(P, [A|X], [A|Y], Z) :- A < P, partition(P, X, Y,  
    ↪ Z).
```

```
partition(P, [A|X], Y, [A|Z]) :- A >= P, partition(P, X, Y,  
    ↪ Z).
```

```
partition(P, [], [], []).
```

Şekil: Prolog hızlı sıralama algoritması



# Prolog Örnek Sorular

- Aşağıdaki işlevleri gerçekleştiren fonksiyonları yazınız:
  - ▶ Listenin ilk elemanını veren yüklem
  - ▶ Listenin son elemanını veren yüklem
  - ▶ Listenin sondan bir önceki elemanını veren yüklem
  - ▶ Listedeki en küçük sayıyı veren yüklem
  - ▶ Listedeki elemanların toplamını veren yüklem
  - ▶ Listedeki elemanların karesini veren yüklem
  - ▶ Listedeki negatif sayıları filtreleyen yüklem
  - ▶ Listede ardışık tekrar eden elemanları eleyen/ayıklayan yüklem
- `ele([1, 1, 2, 2, 2, 1, 1, 3], [1, 2, 1, 3]).`
- <https://www.ic.unicamp.br/~meidanis/courses/mc336/2009s2/prolog/problemas/> adresinde 99 problem ve Prolog çözümleri yer almaktadır.



# Curry: Bir Fonksiyonel Mantıksal Dil

- Fonksiyonel bir dilde, program, verileri başka verilere dönüştürmek için veriler üzerinde işlem yapma kurallarını belirleyen bir dizi işlem tanımıdır
- Mantıksal bir dilde, program, bir problemin çözümü için bir ispatın inşa edildiği bir dizi kural ve gerçektir
- Bunların her birinin belirli dezavantajları vardır
- Curry dili, fonksiyonel ve mantıksal programlamanın avantajlarını tek bir dilde bir araya getiriyor



# Curry'de Fonksiyonel Programlama

- Curry, Haskell'in bir uzantısıdır
  - ▶ Fonksiyonel programlama için Haskell'in sözdizimini ve anlamını korur
  - ▶ Mantıksal programlama için yeni sözdizimi ve semantik ekler
- Fonksiyon tanımları Haskell'deki gibi denklem setleridir
- Curry, tembel değerlendirme kullanır





# Belirsizlik, Koşullar ve Geri İzleme Ekleme

- Saf bir fonksiyonel dil yalnızca deterministik hesaplamayı destekler
  - ▶ Bir fonksiyonun belirli bir argüman kümesine uygulanması her zaman aynı değeri üretir
- Yazı tura atmak gibi sorunlar, çözümleri bir dizi değerden geldiği için yeterince tanımlanmamıştır
- Curry, belirli bir sıra olmadan, bir fonksiyon için bir dizi denklemin ? seçim operatörünü kullanarak denenmesine izin vererek belirsizliği destekler



# Belirsizlik, Koşullar ve Geri İzleme Ekleme

- Örnek:

$$x \text{ ? } y = x$$

$$x \text{ ? } y = y$$

- Curry otomatik olarak ilk denklemi denemez
- Biri başarısız olursa, başka bir denklem denenir
- Yazı tura atmak için belirsiz olan(nondeterministic) bir işlev:

$$\text{flipCoin} = 0 \text{ ? } 1$$


# Belirsizlik, Koşullar ve Geri İzleme Ekleme

- `sorted`, bir listeyi argüman olarak bekler ve aynı öğelerin sıralı bir listesini döndürür:

```
sorted [] = []
```

```
sorted [x] = [x]
```

```
sorted (x:y:xs) = | x <= y
                  = x : sorted(y:ys)
```

- Üçüncü satır, yalnızca listedeki ilk öğe ikinci öğeden küçük veya ona eşitse sağ tarafının değerlendirilmesine izin veren bir koşulu (`|` sembolünün sağında) içerir



# Belirsizlik, Koşullar ve Geri İzleme Ekleme

- `permutation` fonksiyonu, boş olmayan bir listenin ilk öğesini bu listenin geri kalanının bir permütasyonuna ekler

```
permutation [] = []
```

```
permutation (x:xs) = insert x (permutation xs)
```

- `insert` fonksiyonu, bir öğeyi bir listede rastgele bir konuma yerleştirir
  - ▶ Boş olmayan listeler için kesin olmayan(nondeterministically) bir şekilde tanımlanmıştır

```
insert x ys = x:ys
```

```
insert x (y:ys) = y : insert x ys
```



# Mantıksal Değişkenler Ekleme ve Birleştirme

- Mantıksal değişkenler ve birleştirme, Curry'ye bilinmeyen veya kısmi bilgiler içeren denklemleri çözme yeteneği verir
  - ▶ Bazı değişkenleri, onları içeren bir dizi denklemi tatmin edecek şekilde somutlaştırılmaları anlamında bağımsız olarak görmeyi içerir
- Curry, bu şekilde çözülecek denklemi belirtmek için  $==$  sembolünü kullanır
- Örnek: `zs ++ [2] == [1, 2]`



# Curry Örnekleri

- <https://www.informatik.uni-kiel.de/~mh/curry/examples/> adresinden erişilebilir.



# Kaynaklar



Robert Kowalski.

Algorithm = logic+ control.

*Communications of the ACM*, 22(7):424–436, 1979.



Niklaus Wirth.

*Algorithms + data structures = programs.*

Prentice-Hall.

