BMB214 Programlama Dilleri Prensipleri

Ders 3. Sözdizimi (Syntax) ve Anlambilim (Semantics) Tanımlama

Erdinç Uzun
Tekirdağ Namık Kemal Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Giriş
- Sözdizimi Tanımlamadaki Genel Problem
- Sözdizimi Tanımlamanın Biçimsel Yöntemleri
- Öznitelik Grameri
- Programların Anlamlarını Tanımlama: Dinamik Anlamlar
  - İşlevsel Anlambilim
  - Matematiksel Nesnelerle Anlambilim
  - Aksiyomatik Anlambilim

Giriş

- Bir programlama dili geliştirilirken aktörler
  - Tasarımcı/lar ve ekibi
  - Programcı ve Geliştiriciler (Şirket veya topluluk)
    - İlk değerlendirmeler
    - Geri dönüşlere göre iyileştirmeler
  - Programcı ve Geliştiriciler (Tüm dünya)
    - Programlama dilinin popülerliği
    - İyi dokümante edilmesi
    - Üzerine yardımcı kütüphaneler çıkması
      - Açık kaynak kod sistemler
      - Git
    - Geri dönüşler
    - Bakım ve güncelleme

Değerlendirmelerde ön plana çıkan konular

- Okunabilirlik, yazılabilirlik ve güvenilirlik
- Programlama dilinde seçilen sözcükler ve deyimler
- Sözdizimi ve anlambilim

Sözdizimi ve Anlambilim

- Sözdizimi (syntax): Bir program yazarken takip edilmesi gereken zorunlu kurallar dizisidir.
  - Expressions (İfadeler)
  - Statements (Anlatımlar)
  - Blocks
- Anlambilim (semantics), bir program dilindeki bir ifade, anlatım ve blokların ne anlama geldiğidir.
  - Çay -> Dere
  - Çay -> İçilen
- Sözdizimi ve anlambilim bir dilin tanımını verir

Sözdizimi ve Anlambilim

- while(boolean_exp) statement - block
- Bu statement biçiminin anlam bilgisi (semantics), Boole ifadesinin mevcut değeri doğru olduğunda, bloğun çalıştırılmasıdır. Daha sonra kontrol, işlemi tekrarlamak için dolaylı olarak Boole ifadesine döner. Boolean ifadesi yanlışsa, kontrol while yapısını izleyen statement'a aktarılır.
- Sözdizimi ve anlambilim yakından ilişkilidir. İyi tasarlanmış bir programlama dilinde, anlambilim doğrudan sözdiziminden gelmelidir; yani, bir statement'ın görünümü, statement'ın neyi başarmayı amaçladığını güçlü bir şekilde önermelidir.
- Sözdizimini tanımlamak, anlambilimini tanımlamaktan daha kolaydır, çünkü kısmen kısa ve evrensel olarak kabul edilen bir gösterim sözdizimi açıklaması için kullanılabilir, ancak henüz anlambilim için hiçbiri geliştirilmemiştir.

Sözdizimi: Expressions (İfadeler)

- Tek bir değer (value) olarak değerlendirilen bir dizi değişken (variables), operatör ve metot çağrısıdır (dilin sözdizimine göre oluşturulmuştur).
- Görevi iki yönlüdür:
  - Expression'ın öğeleri tarafından belirtilen hesaplamayı gerçekleştirmek.
  - Hesaplamanın sonucu olan bir değeri döndürmek.
- Bir expression örneği: Character.isUpperCase('Deneme')
  - İşlem: isUpperCase metotu çağrılır
  - Sonuç: Metottan dönen değer (true veya false)
- Bir expression örneği:
  - x + y / 100 : Belirsiz
  - (x + y) / 100 : Belirli, tavsiye edilen
  - Görüldüğü gibi bir expression'da işlem önceliği önemlidir.

Sözdizimi: Statements (Anlatımlar)

- Statement, kabaca doğal dillerdeki cümlelere eşdeğerdir. Bir statement, tam bir yürütme birimi oluşturur. Aşağıdaki ifade türleri, ifade noktalı virgülle (;) sonlandırılarak bir ifadeye dönüştürülebilir:
  - Atama ifadeleri:
    - aValue = 8933.234;
  - Herhangi bir ++ veya -- kullanımı:
    - aValue++;
  - Yöntem çağrıları:
    - System.out.println(aValue);
  - Nesne oluşturma ifadeleri:
    - Integer integerObject = new Integer(4);
  - Tanımlama (Declaration) statement
    - double aValue = 8933.234;
  - Akış kontrol (Control flow) statement:
    - if ve while

Sözdizimi: Blocks

- Bloklar, uygun parantezler arasındaki sıfır veya daha fazla statement'tan oluşan bir gruptur ve tek bir statement'a izin verilen her yerde kullanılabilir.
- Aşağıdaki liste, MaxVariablesDemo programından her biri tek bir statement içeren iki bloğu gösterir:

```
if (Character.isUpperCase(aChar)) {
    System.out.println("The character " + aChar + " is upper case.");
} else {
    System.out.println("The character " + aChar + " is lower case.");
}
```

Sözdizimi Tanımlama: Terminoloji

- Bir doğal dilde, **Cümle (sentence)** alfabelerdeki karakterlerden oluşan dizi
  - Programlama tarafında statement
- **Dil (language)** cümleler kümesidir. Örneğin, Türkçe birçok bir cümle tanımlamasında çok miktarda ve kompleks kurallar belirlemek gerekir. Bir programlama dilinin kuralları ile karşılaştırıldığında bir programlama dilini öğrenmek çok daha kolaydır☺
- **Sözcük (lexeme),** programlama dilinde en düşük seviyeli sözdizimsel birim olan bir karakter dizisidir. Bunlar programlama dilinin "kelimeleri" ve noktalama işaretleridir.
  - Identifiers: bir programlama dilindeki değişken, metot, sınıf ve benzeri özelliklerin isimleri denir.
- **Sembol (token),** bir sözcük birim sınıfını oluşturan sözdizimsel bir kategoridir. Bunlar programlama dili için "isimler", "fiiller" ve diğer konuşma parçalarıdır.

Lexeme ve Token Örneği

- while (y >= t) y = y - 3 ;

Lexeme	Token
while	WHILE
(	LPAREN
y	IDENTIFIER
<=	COMPARISON
t	IDENTIFIER
)	RPAREN
y	IDENTIFIER
=	ASSIGNMENT
y	IDENTIFIER
-	ARITHMETIC
3	INTEGER
;	SEMICOLON

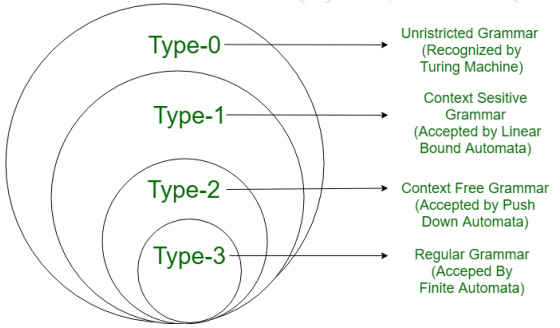
Dillerin Biçimsel Tanımı

Bir dilin tanımı iki yolla yapılır:

- Tanıyıcılar(Recognizers): Bir tanıma aracı dilin girdi dizilerini okur ve girilen diziler o dile ait mi değil mi karar verir. (Compiler)
  - Örnek: bir derleyicinin sözdizimi analizi (syntax analyzer) kısmı
- Üreticiler(Generators): Bir dilin cümlelerini üreten araç– Bir kimse belli bir cümlenin söz diziliminin doğru olup olmadığına onu üreticinin yapısıyla karşılaştırarak karar verebilir. (Grammar, BNF… Bir sonraki bölümde ayrıntıya girilecektir.)

Context -Free Grammars (Bağlamdan Bağımsız Gramerler)

- Noam Chomsky tarafından 1950'lerin ortalarında geliştirildi
  - Dört dil sınıfını tanımlayan dört gramer sınıfı tanımladı
  - Chomsky bir dilbilimci olduğu için, birincil ilgi doğal dillerin teorisi üzerine çalıştı, programlama dilleri ile ilgisi yoktu
- Programlama dillerinde Type-3 ve Type-2 ön plana çıkar.
  - Programlama dillerindeki token formları Type-3 – düzenli (regular) gramer sınıfına girer.
  - Küçük istisnalar dışında programlama dillerinin sözdizimi Type-2 – Bağlamdan Bağımsız Gramer sınıfındadır.
- Dil üreticileri, doğal dillerin söz dizimini tanımlama için kullandılar



Backus -NaurForm (BNF) - 1959

- John **Backus** tarafından Algol58'i tanımlamak için geliştirildi.
- Yeni gösterim daha sonra ALGOL 60'ın açıklaması için Peter **Naur** tarafından biraz değiştirildi.
- BNF bağlamdan bağımsız gramere denktir
- BNF diğer dilleri tanımlamada kullanılan bir metadildir.
  - Bir metadil, başka bir dili tanımlamak için kullanılan bir dildir.

BNF Soyutlama

- BNF'de söz dizimsel yapıların sınıflarını göstermek için **soyutlamalar (abstractions)** kullanılır. Bunlar sözdizimsel değişkenler gibi davranırlar
- Örneğin bir atama işlemi için
  - <assign>: küçük büyük işaretleri soyutlamaların ismini sınırlamak için kullanılır.
  - <assign> → <var> = <expression>
  - Ok işaretinin sol tarafındaki metin left-hand side (LHS) olarak isimlendirilir ve soyutlamayı tanımlar.
  - Sağ taraf ise (right-hand side - RHS), diğer soyutlamalar için tokens, lexems ve referenaslar içerir.
  - Atama bir değişken <var>, eşittir '=' ve bir ifadeden <expression> oluşur.

BNF Atama örneği

- C, Java, C#, Javascript gibi bir dilde atama
  - degisken = x + y
- Soyutlamalar non-terminal iken Lexeme ve token kuralları ise terminal olarak isimlendirilir.
- BNF tanımı (gramer) kurallar topluluğudur.

Farklı programlama dillerinde atama işlemi

=	Awk, B, Basic, BourneShell, C, C#, C++, Classic REXX, Erlang, Go, Icon, Io, Java, JavaScript, Lua, Mathematica, Matlab, Oz, Perl, Perl6, PHP, Pike, YCP, Yorick
:=	Ada, BCPL, Cecil, Dylan, E, Eiffel, Maple, Mathematica, Modula-3, Pascal, Pliant, Sather, Simula, Smalltalk, SML
<-	F#, OCaml
_	Squeak
:	BCPL, Rebol
->	Beta
Def	PostScript
setq	Common Lisp, Emacs Lisp
setf	Common Lisp
set	Common Lisp, FishShell, Rebol
SET v=...	MUMPS
set!	Scheme
is	Prolog
make "v e	Logo
e v !	Forth

Çoklu kural

- Örneğin if statement
  - <if_stmt> → if (<logic_expr>) <stmt>
  - <if_stmt> → if (<logic_expr>) <stmt> else <stmt>
- Görüldüğü gibi if ifadesi için iki kural vardır. Bu durumda | 'OR' ile iki statement bağlanabilir.
  - **<if_stmt>** → if (<logic_expr>) <stmt>
    | if (<logic_expr>) <stmt> else <stmt>
- <stmt> bir satırı ifade eder. (Programlama diline göre birden fazla satır | şeklinde yazılmaya devam edilebilir.)
  - **<if_stmt>** → if (<logic_expr>) { <stmt_list> }

If statement – Java vs. Python

Java

```
if (x > 3) {
    x -= 2;
    System.out.println(x);
}
y = x;
```

Python

```
if x > 3:
    x -= 2
    print x
y = x
```

Java

```
if (x > 3) {
    x -= 2;
    System.out.println(x);
}
else
    System.out.println('Error');
y = x;
```

Python

```
if x > 3:
    x -= 2
    print(x)
else:
    print('Error')
y = x
```

- If statement için birden fazla satır olunca Java'da parantez var iken Python'da yoktur.
- Java'da karşılaştırma işlemi parantezler arasına yazılır.
- Python'da şarttan sonra blok bölümü başlarken ':' kullanılır.
- Görüldüğü gibi söz dizimleri arasında ufak farklılıklar olabilir.
- Bir önceki BNF sözdizimi hangi programlama diline uygundur? Bu BNF sözdizimi Java hariç hangi programlama dillerinde kullanılır?
- BNF basit olmasına rağmen, neredeyse tüm programlama dilleri sözdizimini tanımlamak için yeterince güçlüdür.

BNF ile Listelerin tanımı

- Matematikte (. . .) veya 1, 2, … şeklinde kullanımlar vardır.
- BNF'de özyineleme (recursion) kullanılır. Kuralı:
  - LHS'si RHS'sinde görünüyorsa yinelemelidir.
- <ident_list>→ identifier

  | identifier, <ident_list>
- Liste için burada virgül kullanılmıştır.

Gramer ve Derivasyon (Grammers and Derivations)

- Gramer dilleri tanımlamak için üretken bir araçtır.
- Dilin cümleleri, **başlangıç sembolü (start symbol)** başlayarak, kuralların bir dizi uygulamasıyla oluşturulur. Bu kural uygulamaları dizisine derivasyon adı verilir.
- Tam bir programlama dili için bir gramerde, başlangıç sembolü tam bir programı temsil eder ve genellikle <program> olarak adlandırılır.

Örnek bir gramer: Örnek - 1

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

- stmts: Komutlar, program içindeki komut satırları
- stmt: Bir tek komut, bir tek komut satırı
- var: Değişken
- expr: İfade
- term: terim
- const: Sabit (3 veya 17 gibi)

Derivasyon kodu (Örnek - 1)

Gramer

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Derivasyon

```
<program> => <stmts> => <stmt>
=> <var> = <expr> => a =<expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const
```

Tek satırlık kod

var olarak a seçildi

İlk exp seçildi

term olarak var seçildi

var b seçildi

term olarak const seçildi

Örnek gramerde, seçimler yapılarak derivasyon yapılmıştır.

Bir gramer örneği: (Örnek - 2)

<program> → **begin** <stmt_list> **end**

<stmt_list> → <stmt> | <stmt> ; <stmt_list>

<stmt> → <var> = <expression> <var> →A|B|C

<expression> → <var> + <var> | <var> - <var> | <var>

- «begin» ve «end» özel kelimeleri blok oluşturmak için kullanılmış.
- «stmt» sonuna ; geliyor
- A, B, C değişkenlerinde «var» biri olabilir
- «expression» üç farklı durumda olur. İki durum + ve - operatörlerini içerir.

Derivasyonu (Örnek - 2)

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt> → <var> = <expression> <var> →A|B|C
<expression> → <var> + <var> | <var> - <var> | <var>

<program> => **begin** <stmt_list> **end**
=> **begin** <stmt>;<stmt_list>**end**
=> **begin** <var>=<expression>;<stmt_list>**end**
=> **begin** A = <expression> ; <stmt_list> **end**
=> **begin** A = <var> + <var> ; <stmt_list> **end**
=> **begin** A = B + <var> ; <stmt_list> **end**
=> **begin** A = B + C ; <stmt_list> **end**
=> **begin** A = B +C ;<stmt>**end**
=> **begin** A = B + C ; <var> = <expression> **end**
=> **begin** A = B +C ;B =<expression>**end**
=> **begin** A = B +C ;B =<var>**end**
=> **begin** A =B +C ;B = C **end**

Tüm durumların derivasyonu yapılmıştır.

Gramer: Örnek - 3

<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <id> + <expr> | <id> * <expr> |(<expr>)| <id>

- A = B * (A + C)

Derivasyon: Örnek - 3

<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <id> + <expr> | <id> * <expr> |(<expr>)| <id>

- A = B * (A + C)
  - Soldan başlayarak derivasyon yapıldığı durumda

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = B * <expr>

=> A = B * (<expr>)

=> A = B * (<id> + <expr>)

=> A = B * (A + <expr>)

=> A = B * (A + <id>)

=> A = B * (A + C)

Gramer Ayrıştırma Ağaçları (Parse Trees)

- Gramerlerin en önemli özelliklerinden biri, tanımladıkları dillerin cümlelerinin hiyerarşik sözdizimsel yapısını doğal olarak tanımlamalarıdır
- Bu hiyerarşik yapılara ayrıştırma ağaçları denir
- Ayrıştırma ağaçları düğümlerden, dallardan ve yapraklardan oluşur
- Ayrıştırma ağacının her düğümü bir non-terminal sembolle etiketlenir
- Her yaprak bir terminal simgesi ile etiketlenir
- Ayrıştırma ağacının her alt ağacı, cümledeki bir soyutlamanın bir örneğini açıklar

Örnek - 1 için Ayrıştırma Ağacı

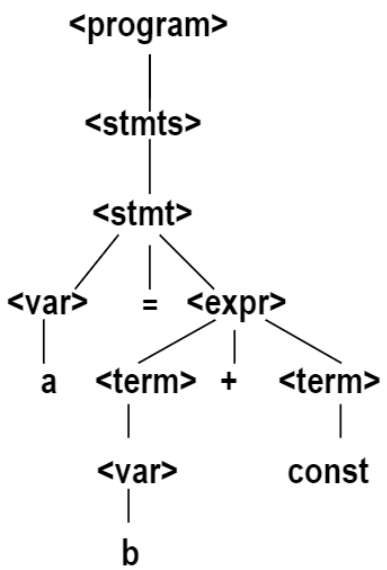
Gramer

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<var> → a | b | c | d
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
```

Derivasyon

```
<program> => <stmts> => <stmt>
          => <var> = <expr> => a =<expr>
          => a = <term> + <term>
          => a = <var> + <term>
          => a = b + <term>
          => a = b + const
```

Tree



Örnek - 3 için Ayrıştırma Ağacı

<assign> → <id> = <expr>
<id> → A|B|C
<expr> → <id> + <expr> | <id> * <expr> |(<expr>)| <id>

A = B * (A + C)

<assign> => <id> = <expr>

=> A = <expr>

=> A = <id> * <expr>

=> A = B * <expr>

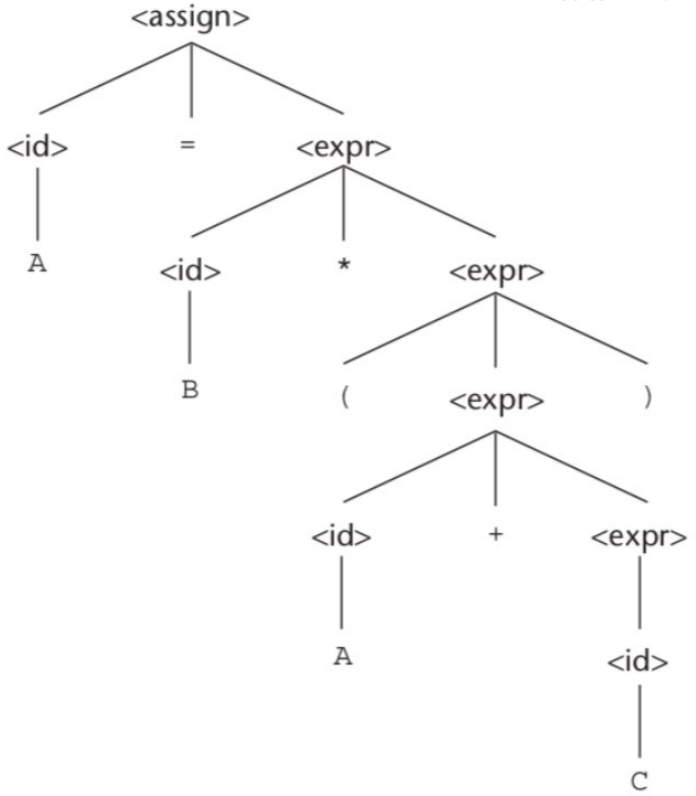
=> A = B * (<expr>)

=> A = B * (<id> + <expr>)

=> A = B * (A + <expr>)

=> A = B * (A + <id>)

=> A = B * (A + C)



Gramerlerde Belirsizlik (Ambiguity)

- İki veya daha fazla ayrı ayrıştırma ağacına sahip bir cümlecik formu üreten bir gramer belirsizdir
- Belirsizlik bir ifadenin birden fazla anlama gelmesi durumunda söz konusu olur
- Gramerde belirsizliği tespit etmek için bir cümle iki ayrı ayrıştırma ağacı ile oluşturulmaya çalışılır.
- Eğer oluşturulabilirse gramer belirsizdir

Gramerlerde Belirsizlik (Ambiguity)

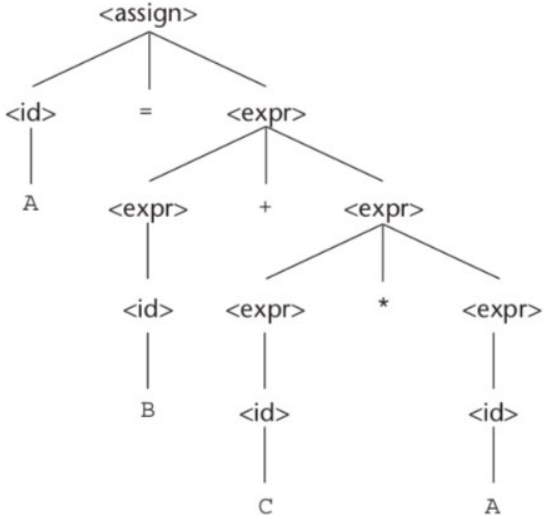
<assign> → <id> = <expr>

<id> →A|B|C

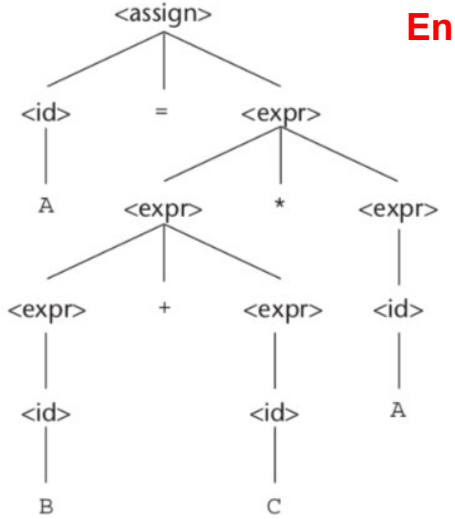
<expr> → <expr>+<expr> | <expr>*<expr> |(<expr>) | <id>

A = B + C * A

Sol taraflı derivasyon



Sağ taraflı derivasyon



İki ağaç durumu var,
En önemlisi sonuçlar farklı

Operatörlerin Önceliği: Gramerde belirsizliği ortadan kaldırma

A = B + C * A

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <expr>

| <expr> * <expr>

| (<expr>)

| <id>

Belirsiz durum

<assign> → <id> = <expr>

<id> → A | B | C

<expr> → <expr> + <term>

| <term>

<term> → <term> * <factor>

| <factor>

<factor> → (<expr>)

| <id>

Belirsizlik ortadan kaldırıldı

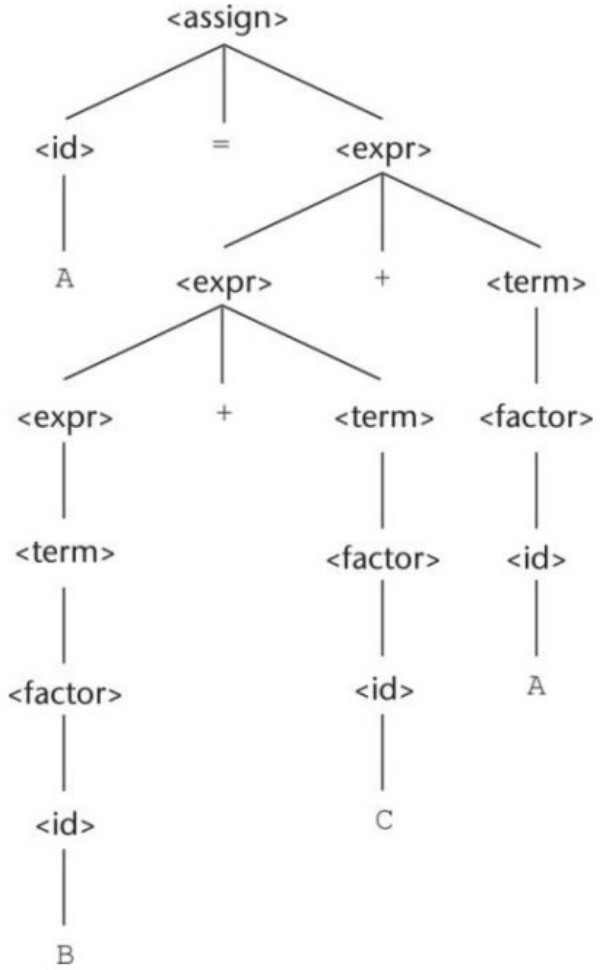
- Bu değişiklik sayesinde hem soldan hem de sağdan aynı ayrıştırma ağacı elde edilir.
- Derivasyon ve ağaç çizimini size bırakıyorum ☺ Kolay gelsin…

Operatörlerin İlişkilendirilmesi

- Operatör önceliği yok ise başka bir deyişle operatörler aynı öncelikte ise bir anlamsal kurala ihtiyaç vardır. Örneğin:
  - A / B * C
  - A / B / C
  - A = B + C + A

Operatörlerin İlişkilendirilmesi: A = B + C +A

- Sol taraflı bir derivasyon ile yandaki ayrıştırma ağacı elde edilir.
- Sol veya sağ taraflı olması sadece toplama önceğini değiştirir. Yani:
  - (A + B) + C = A + (B + C) arasında fark yoktur.
- Fakat, bu durumun dahi kayan noktalı sayılarda sonuçların farklı çıkmasına sebebiyet verebileceğini unutmayın.
  - Bazen programlama dilleri aynı işlemde ufak farklılıklar verebilir. Bunun sebebi sol veya sağ taraflı çalışma prensibidir.
- Programlama Dillerinde operatörlere göre yön değişebilir. Çoğu dilde üs alma operatörü sağ taraflıdır.



if – else için Belirsiz Gramer

<if_stmt> → **if** (<logic_expr>) <stmt>

if (<logic_expr>) <stmt> **else** <stmt>

- İç içe if kullanılırsa bu BNF grameri sorun yaratır.
- Sol ve sağ taraflı oluşturulmuş ağaçlar aşağıdaki gibidir
- Program kodunda parantez kullanarak bu belirsizlik çözülebilir

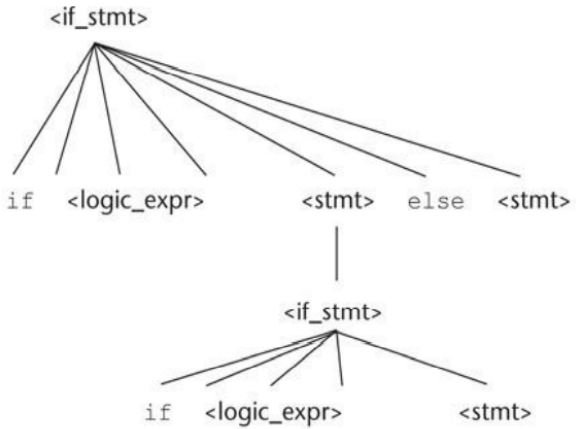
Bir Java veya C# Program Kodu

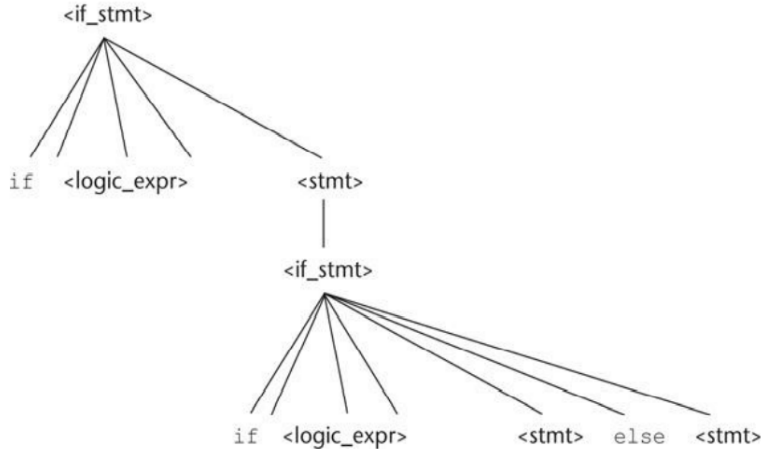
```
if (done == true)

if (denom == 0)

    quotient = 0;

    else quotient = num / denom;
```





Genişletilmiş (Extended) BNF (EBNF)

- EBNF, BNF'nin tanımlayıcı gücünü artırmaz; sadece okunabilirliğini ve yazılabilirliğini arttırırlar.

<if_stmt> → **if** (<expression>) <statement> [**else** <statement>]

Durum 1: EBNF'de, köşeli parantezler seçimlik durumu belirtir. Kod sağ taraflı derivasyona (RHS) uygulanmalıd
Bu kodun BNF gösterimi:

<if_stmt> → **if** (<expression>) <statement>

|**if** (<expression>) <statement> **else** <statement>

EBNF

- Durum 2: Tekrarlar (0 veya daha fazla tekrar) kıvırcık parantez {} içine yerleştirilir

<ident_list> → <identifier> {, <identifier>}

- Durum 3: Sağ tarafların alternatif kısımları normal parantezler arasına dikey çizgi (OR) ile ayrılarak yerleştirilir

EBNF

<term> → <term> (* | / | %) <factor>

BNF

<term> → <term> * <factor>

| <term> / <factor>

| <term> % <factor>

BNF ve EBNF Örnekleri

BNF

```
<expr> → <expr>+<term> | <expr>-<term> | <term>
<term> → <term> * <factor> | <term> / <factor>
                | <factor>
```

EBNF

```
<expr> → <term> { (+ | -) <term>}
<term> → <factor> { (* | /) <factor>
```

BNF

```
<expr> → <expr> + <term> | <expr> - <term> | <term>
<term> → <term> * <factor> | <term> / <factor> | <factor>
<factor> → <exp> ** <factor> <exp>
<exp> → (<expr>) |id
```

EBNF

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → (<expr>) |id
```

Öznitelik Grameri

- Bir öznitelik grameri, bağlamdan bağımsız gramerler ile tanımlanamayan bir programlama dilinin yapısını tanımlamak için kullanılan bir araçtır
- Bir öznitelik grameri, bağlamdan bağımsız bir dilbilgisinin bir uzantısıdır
- Uzantı, tür uyumluluğu gibi belirli dil kurallarının uygun şekilde tanımlanmasına izin verir
- Öznitelik gramerini tanımlayabilmek için **statik anlambilim** ve **statik anlam kontrolü (derleyici tasarımı)** kavramlarının tanımına ihtiyaç vardır
- Ayrıştırma ağaçları boyunca bazı anlamsal bilgileri taşımak için bağlamdan bağımsız gramerlere eklemeler yapılır

Statik Anlambilim

- BNF ile bir dilin tüm yapısını tanımlamak zordur hatta imkansızdır
- Örneğin, tür kontrolü konusunda; Java'da, bir floating-point değer bir integer değişkene atanamaz.
- Bu kısıtlama BNF'de belirtilmesine rağmen, ek non-terminal semboller ve kurallar gerektirir
- Java'nın tüm yazım kuralları BNF'de belirtilmiş olsaydı, gramerin boyutu çok büyük olurdu
- Sorun yaratan yapı kategorileri:
  - Bağlamdan bağımsız, ancak kullanışsız (örneğin, bir expression'daki operand'ın tipi)
  - Bağlamdan bağımsız olmayan durumlar (örneğin, değişken tanımlamadan önce bildirilmelisi)
- BNF ile tanımlanamayan veya tanımlanması çok zor olan kurallar, statik semantik kuralları adı verilen dil kuralları kategorisindedir
- Bir dilin statik semantiği, yürütme sırasındaki programların anlamıyla doğrudan doğruya ilişkilidir

Statik Anlambilim

- Bir dildeki birçok statik anlamsal kural, tür kısıtlamalarını belirtir
- Bu kuralların analizi derleme zamanında yapıldığı için adına Statik semantik denmiştir
- BNF ile statik anlambilim tanımlayamama problemleri nedeniyle, bu görev için çeşitli daha güçlü mekanizmalar tasarlanmıştır
- Böyle bir mekanizma olan öznitelik grameri, Knuth tarafından hem programların sözdizimini hem de statik semantiği tanımlamak üzere tasarlanmıştır
- Öznitelik grameri, bir programın statik semantik kurallarının doğruluğunu tanımlamak ve kontrol etmek için resmi bir yaklaşımdır
- Derleyici tasarımında her zaman resmi bir biçimde kullanılmasalar da, öznitelik gramerinin temel kavramları en azından her derleyicide gayri resmi olarak kullanılır
- Deyimlerin, ifadelerin ve program birimlerinin anlamı olan sematiktürü dinamik semantiktir

Öznitelik Gramerleri

- Öznitelik grameri niteliklere, nitelik hesaplama fonksiyonlarına ve doğrulama fonksiyonlarına (attributes, attribute computation functions, and predicate functions) eklenen **bağlamdan bağımsız** gramerlerdir
- Gramer simgeleriyle (terminal ve nonterminal simgeler) ilişkilendirilen nitelikler kendilerine atanmış değerlere sahip olabilecekleri şekilde değişkenlere benzer niteliktedir.
- **Nitelik hesaplama fonksiyonları**, bazen anlamsal fonksiyonlar olarak adlandırılır ve gramer kurallarıyla ilişkilendirilir
  - Özellik değerlerinin nasıl hesaplandığını belirtmek için kullanılırlar
- Dilin statik semantik kurallarını belirleyen **doğrulama fonksiyonları** gramer kurallarıyla ilişkilendirilir

Öznitelik Gramerleri: Tanım
Nitelik Hesaplama Fonksiyonları

- Bir nitelik grameri, aşağıdaki ek özelliklere sahip bir gramerdir
  - Her bir gramer sembolü X ile ilişkili bir dizi nitelik A(X)'dir. A(X) kümesi, sırasıyla **sentezlenmiş** ve **miras alınan öznitelikler** olarak adlandırılan iki ayrık kümeden S (X) ve I (X) oluşur.
  - **Sentezlenen öznitelikler – S(X)**, anlamsal bilgileri bir ayrıştırma ağacına geçirmek için kullanılırken, **miras alınan öznitelikler – I(X)** anlamsal bilgileri aşağıya ve bir ağaç boyunca iletir.
  - Her bir kuralın kuraldaki non-terminallerin belli niteliklerini tanımlayan fonksiyonlar kümesi vardır
  - Her kuralın, niteliklerin tutarlılıklarını kontrol etmek üzere belirteçler kümesi vardır

Öznitelik Gramerleri: Tanım…
Sentezlenmiş Nitelik ve Miras Alınan Öznitelik

- Her gramer kuralı ile ilişkili olarak, gramer kuralındaki simgelerin nitelikleri üzerinde bir dizi **semantik fonksiyon** ve muhtemelen boş bir **doğrulama fonksiyonu** kümesi vardır
- X0 → X1…Xn bir kural için, X0'ın sentezlenmiş niteliği S(X0)=f(A(X1), …, A(Xn)) şeklinde semantik fonksiyonlar ile hesaplanır.
  - Başka bir deyişle, bir ayrıştırma ağacı düğümündeki sentezlenmiş bir özniteliğin değeri - S(X0), yalnızca o düğümün alt düğümlerindeki özniteliklerin değerlerine bağlıdır.
- Miras nitelikleri sembolleri Xj, 1≤j≤n olmak üzere I(Xj=f(A(X0), …, A(Xn)) şeklinde bir semantik fonksiyon ile hesaplanır.
- Ayrıştırma ağacı düğümünde (nodes) kalıtsal bir öznitelik değeri, o düğümün kendine özgü düğümünün ve onun kardeş düğümlerinin özellik değerlerine bağlıdır. Döngüselliğin önüne geçmek için kalıtsal nitelikler genellikle I(Xj) = f (A(X0), …, A (X (j-1)) şeklinde fonksiyonlar ile sınırlandırılmıştır.
- Bu form, miras alınmış bir özniteliğin kendisine veya ayrıştırma ağacındaki sağdaki niteliklere bağlı kalmasını önler

Öznitelik Gramerleri: Tanım

- Bir **doğrulama fonksiyonu**, {A(X0), …, A(Xn)} öznitelikleri birleşimi ve bir literal öznitelik değerleri kümesi üzerinde bir Boolean ifadesi biçimindedir.
- Bir öznitelik grameri ile izin verilen tek bir derivasyon (her nonterminal bir doğrulama fonksiyonu ile ilişkilidir) doğrudur (true). Sonucu yanlış (false) olan bir doğrulama fonksiyonu, sözdiziminin veya dilin statik anlam kuralının ihlal edildiğini gösterir.
- Öznitelik gramerinin bir ayrıştırma ağacı, muhtemelen boş olan her düğüme eklenen öznitelik değerleri kümesiyle birlikte, alttaki BNF dilbilgisine dayanan ayrıştırma ağacından oluşur
- Ayrıştırma ağacındaki tüm öznitelik değerleri hesaplandıysa, ağacın tamamen özelliklendirildiği söylenir
- Uygulamada her zaman bu şekilde yapılmamasına rağmen, öznitelik değerlerinin, özelliklendirilmemiş ayrıştırma ağacının **derleyici** tarafından oluşturulduktan sonra hesaplandığı düşünülür.

Yapısal Nitelikler (Intrinsic Attributes)

- Yapısal öznitelikler, değerleri ayrıştırma ağacının dışında belirlenen yaprak düğümlerden sentezlenen özniteliklerdir
- Örneğin, bir programdaki değişken örneğinin türü, değişken adlarını ve türlerini depolamak için kullanılan sembol tablosundan gelebilir
- Sembol tablosunun içeriği, daha önceki deklarasyon ifadelerine dayanarak belirlenir. Başlangıçta, özelliklendirilmemiş ayrıştırma ağacının yapılandırıldığını ve öznitelik değerlerinin gerekli olduğunu varsayarak, değerleri olan tek öznitelik, yaprak düğümlerinin öz nitelikleridir
- Bir ayrıştırma ağacındaki yapısal öznitelik değerleri göz önüne alındığında, semantik fonksiyonlar kalan öznitelik değerlerini hesaplamak için kullanılabilir

Öznitelik Grameri : Bir örnek

- Basit bir atama ifadesinin tür kurallarını denetlemek için bir öznitelik gramerinin nasıl kullanılacağını bu örnek ile inceleyeceğiz.
- Bu atama ifadesinin sözdizimi ve statik semantiği şöyledir:
  - Değişken adları A, B ve C'dir
  - Atamaların sağ tarafı, bir değişken veya bir değişken eklenmiş bir ifade olabilir
  - Değişkenler iki türde olabilir: int veya real
  - Bir atamanın sağ tarafında iki değişken olduğunda, aynı tür olması gerekmez
  - İşlenen türleri aynı olmadığında, ifade türü her zaman real olur
  - Değişkenler aynı olduğunda ifade tipi, işlenenlerin türüdür
  - Atamanın sol tarafının türü, sağ tarafın türüne uygun olmalıdır. Böylece sağ taraftaki işlenenlerin türleri karışık olabilir
  - Ancak atama yalnızca hedef ve sağ tarafın değerlendirilmesinden elde edilen sonucun değeriyle aynı türde olması durumunda geçerlidir
  - Öznitelik grameri bu statik anlamsal kuralları belirtir

Öznitelik Grameri : Bir örnek…

- Söz dizimi
  <assign> → <var> = <expr>
  <expr> → <var> + <var> | <var>
  <var> → A | B | C
- Actual_type (gerçek_tür): <var> ve <expr> için sentezlenmiştir. Bunlar <var> veya <expr> de int ve real gerçek türlerini depolamak için kullanılır. var içinde saklanırsa, **gerçek tür yapısaldır**. expr içinde saklanırsa çocuk düğümlerin **gerçek türlerinden** belirlenir
- Expected_type(beklenen_tür): <expr>için mirastır. Atama ifadesinin sol tarafındaki var'ın türü kullanılarak belirlenir

Öznitelik Grameri : Bir örnek …
Kurallar belirleniyor…

1. Syntax rule: <assign> → <var> = <expr>

   Semantic rule: <expr>.expected_type ← <var>.actual_type

2. Syntax rule: <expr> → <var>[2] + <var>[3]

   Semantic rule: <expr>.actual_type ←

   if (<var>[2].actual_type = int) and

   (<var>[3].actual_type = int)

   then int

   else real

   end if

   Predicate: <expr>.actual_type == <expr>.expected_type

3. Syntax rule: <expr> → <var>

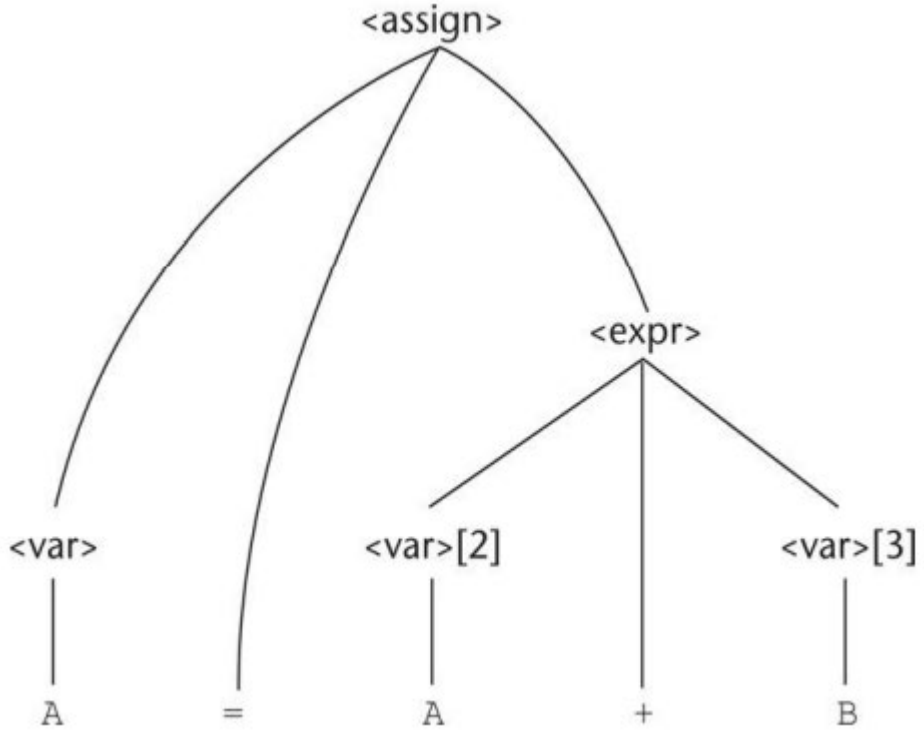
   Semantic rule: <expr>.actual_type ← <var>.actual_type

   Predicate: <expr>.actual_type == <expr>.expected_type

4. Syntax rule: <var> → `A | B | C`

   Semantic rule: <var>.actual_type ← look-up(<var>.string)

Öznitelik Grameri : Bir örnek…
A = A + B için Ayrıştırma Ağacı

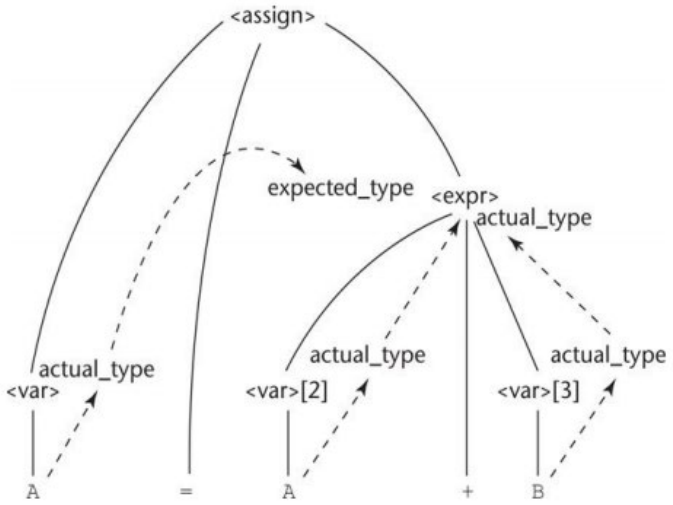


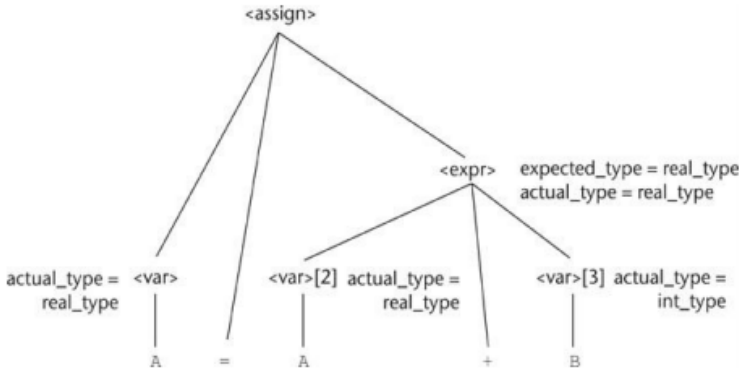
Öznitelik Grameri : Bir örnek…
A = A + B için Ayrıştırma Ağacı

1. <var>.actual_type ← look-up(A) (Rule 4)
2. <expr>.expected_type ← <var>.actual_type (Rule 1)
3. <var>[2].actual_type ← look-up(A) (Rule 4)

   <var>[3].actual_type ← look-up(B) (Rule 4)
4. <expr>.actual_type ← either int or real (Rule 2)
5. <expr>.expected_type == <expr>.actual_type is either

   TRUE or FALSE (Rule 2)





Programların Anlamlarını Tanımlama: Dinamik Anlamlar

- Anlamı tanımlamanın geniş ölçekte kabul edilen, tek bir notasyonu yoktur
- Bir programlama dilinin anlam (semantic) kuralları, bir dilde sözdizimsel olarak geçerli olan herhangi bir programın anlamını belirler
- Anlamsal tanımlama için var olan yöntemler oldukça karmaşıktır ve hiçbir yöntem söz dizim tanımlamak için kullanılan BNF gibi yaygın kullanıma ulaşmamıştır.

İşlevsel Anlambilim (Operational Semantics)

- Programın anlamını gerçek olarak ya da simülasyonla, ifadelerini makine üzerinde çalıştırarak tanımlar. Makinenin (memory, register, vs.) durumundaki değişim ifadenin anlamını tanımlar.
- Yüksek seviye bir dil için işlevsel anlambilimi kullanmak bir sanal makine gerektirir
  - Donanımsal bir saf yorumlayıcı çok pahalı olurdu
- Yazılımsal bir saf yorumlayıcının da problemleri var
  - Özel bir bilgisayarın detaylı karakteristiği eylemlerin anlaşılmasını güçleştirebilirdi
  - Böyle bir anlambilim tanımlaması makine bağımlı olurdu

İşlevsel Anlambilim…

- Daha iyi bir alternatif: tam bir bilgisayar simülasyonu
- Süreç:
  - Bir çevirici kur (kaynak kodunu ideal bir bilgisayarın makine koduna çeviren)
  - İdeal bir bilgisayar için bir simülatör yap
- İşlevsel anlambilimin değerlendirmesi:
  - Eğer gayri resmi kullanılıyorsa (dil kılavuzu, vs) iyi
  - Eğer resmi olarak kullanılıyorsa (örneğin, VDL) oldukça karmaşık, o PL/I'ın anlambilimini tanımlamada kullanılırdı

İşlevsel Anlambilim…
Dil kılavuzu

C Statement	*Meaning*

`for (expr1; expr2; expr3) {`
`  ...`
`}`
	`  expr1;`
	`loop: if expr2 == 0 goto out`
	`  ...`
	`  expr3;`
	`  goto loop`
	`out:  ...`

- Bir C döngüsünün anlambilim açısından nasıl anlatıldığını yanda görüyorsunuz.
- Sanal makineyi yandaki kodu okuyan bir insan gibi varsayın.

Matematiksel Nesnelerle Anlambilim (Denotational Semantics)

- Özyinelemeli fonksiyon teorisine dayalı
- En soyut anlamsal açıklama yöntemi
- Başlangıçta Scott ve Strachey (1970) tarafından geliştirilmiştir
- Bir dil için bir gösterge belirtimi oluşturma süreci:
  - Her dil varlığı için matematiksel bir nesne tanımlanır
  - Dil varlıklarının örneklerini karşılık gelen matematiksel nesnelerin örnekleriyle eşleyen bir fonksiyon tanımlanır
- Dil yapılarının anlamı yalnızca program değişkenlerinin değerleri ile tanımlanır

Matematiksel Nesnelerle Anlambilim: Program Durumu

- Bir programın durumu, tüm mevcut değişkenlerinin değerleridir
  - $s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$
- VARMAP, bir değişken adı ve bir durum verildiğinde değişkenin geçerli değerini döndüren bir fonksiyon olsun
  - $\text{VARMAP}(i_j, s) = v_j$

Matematiksel Nesnelerle Anlambilim Örneği: Decimal Sayılar

```
<dec_num> →  '0' | '1' | '2' | '3' | '4' | '5' |
             '6' | '7' | '8' | '9' |
             <dec_num> ('0' | '1' | '2' | '3' |
                        '4' | '5' | '6' | '7' |
                        '8' | '9')
```

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9

M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)

M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1

…

M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Matematiksel Nesnelerle Örneği…
Expressions

- Expression'ları Z ∪ {error} eşleyin.
  - Z tam sayılar kümesi olsun ve hata değeri 'error' olsun.
- Expression'ların ondalık sayılar, değişkenler veya bir aritmetik operatörü ve her biri bir ifade olabilen iki işlenen içeren ikili ifadeler olduğunu varsayın.

<expr> → <dec_num> | <var> | <binary_expr>

<binary_expr> → <left_expr> <operator> <right_expr>

<left_expr> → <dec_num> | <var>

<right_expr> → <dec_num> | <var>

<operator> → + | *

Δ: Matematiksel fonksiyon için

Matematiksel Nesnelerle Örneği… Expressions

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
                OR Me(<binary_expr>.<right_expr>, s) =
                                undef)
              then error
           else
           if (<binary_expr>.<operator> == '+' then
              Me(<binary_expr>.<left_expr>, s) +
                     Me(<binary_expr>.<right_expr>, s)
           else Me(<binary_expr>.<left_expr>, s) *
               Me(<binary_expr>.<right_expr>, s)
        ...
```

Matematiksel Nesnelerle Anlambilim: Değerlendirme

- Programların doğruluğunu kanıtlamak için kullanılabilir
- Programlar hakkında düşünmek için titiz bir yol sağlar
- Dil tasarımına yardımcı olabilir
- Derleyici oluşturma sistemlerinde kullanılmıştır
- Karmaşıklığı nedeniyle, dil kullanıcıları için çok az faydalıdır.

Aksiyomatik (Belitsel) Anlambilim (Axiomatic Semantics)

- Biçimsel mantık tabanlı (analize dayalı)
- Asıl amaç: biçimsel program doğrulaması
- Aksiyomlar ya da çıkarım kuralları dildeki her bir statement için tanımlanmıştır (expression'ların başka expression'lara dönüşümüne izin vermek için)
- Mantıksal expression'lara bildirim (assertion) denir.

Aksiyomatik Anlambilim

- Bir statement'ın önündeki bir bildirim (önşart(precondition)), çalıştırıldığı zaman değişkenler arasında true olan ilişki ve kısıtları(constraints) belirtir
- Bir statement'ın arkasından gelen iddiaya son şart (postcondition) denir
- En zayıf ön şart (weakest precondition), son şartı garanti eden asgari kısıtlayıcı önşarttır

Aksiyomatik Anlamın Biçimi

- Pre-, post form: {P} statement {Q}

- Bir örnek: a = b + 1 {a > 1}
  - Mümkün bir önşart: {b > 10}
  - En zayıf önşart: {b > 0}
    - En az değere sahip kısıtlayıcı…

Program İspat Süreci

- Tüm program için son koşul, istenen sonuçtur
- Program boyunca ilk statement'a kadar geri dönün. İlk ifadedeki ön koşul program belirtimiyle aynıysa, program doğrudur.

Aksiyomatik Anlambilim Değerlendirme

- Bir dildeki tüm ifadeler için aksiyomlar veya çıkarım kuralları geliştirmek zordur
- Doğruluk kanıtları için iyi bir araçtır ve programlar hakkında akıl yürütmek için mükemmel bir çerçevedir, ancak dil kullanıcıları ve derleyici yazarları için o kadar yararlı değildir.
- Bir programlama dilinin anlamını açıklamadaki faydası, dil kullanıcıları veya derleyici yazarları için sınırlıdır.

Matematiksel Nesnelerle Anlambilim vs. İşlevsel Anlambilim

- İşlevsel anlambilimde, durum değişiklikleri kodlanmış algoritmalarla tanımlanır.
- Matematiksel Nesnelerle anlambilimde, durum değişiklikleri titiz matematiksel fonksiyonlarla tanımlanır.