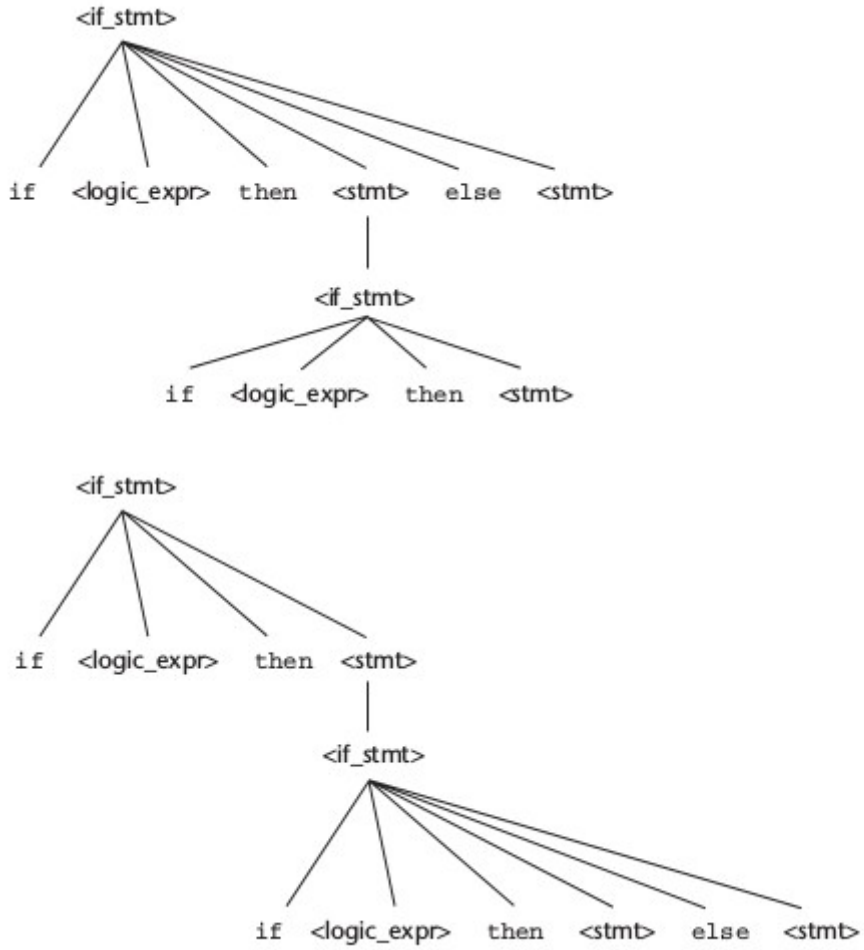
2.1.9. If-then-else için belirsiz bir gramer

ADA dilinde if-then-else ifadeleri için BNF kuralları şöyledir:

<if_stmt> → **if** (<logic_expr>) **then** <stmt>
| **if** (<logic_expr>) **then** <stmt> **else** <stmt>

Eğer <stmt> → <if_stmt> kuralı da varsa bu gramer belirsiz olur. Bu belirsizliği gösteren en basit cümlesel biçim:

if (<logic_expr>) then if (<logic_expr>) then <stmt> else <stmt>
Bu cümlesel biçime ait iki farklı ayrıştırma ağacı **Şekil 1**'deki gibi olabilir.



Şekil 1. if (<logic_expr>) then if (<logic_expr>) then <stmt> else <stmt> cümlesel biçimi için iki farklı ayrıştırma ağacı

Bu yapıdaki bir kod örneğini inceleyelim:

if done==true
then if denom == 0
then quotient = 0
else quotient = num/ denom;

Buradaki sorun else'in hangi if'e ait olduğudur. İlk ayrıştırma ağacında else, if done==true ifadesine aittir. İkinci ağaçta ise else, if denom==0 ifadesine aittir ve yapılmak istenen de budur.

Şimdi if ifadesini doğru tanımlayan belirli bir gramer geliştireceğiz. If yapıları için kural çoğu dilde şudur: else kullanıldıysa o else kendisine en yakındaki eşleştirilmemiş then ile eşleştirilir. Böylece else'i olan bir if in then'i ile else'i arasında, else'i olmayan bir if olamaz. Bir önceki grameri belirli hale getirelim:

<stmt> → <matched> | <unmatched>
<matched> → **if** (<logic_expr>) **then** <matched> **else** <matched>
| any non if statement
<unmatched> → **if** (<logic_expr>) **then** <stmt>
| **if** (<logic_expr>) **then** <matched> **else** <unmatched>

Yukarıdaki grameri kullanarak oluşturulan,
if (<logic_expr>) **then** **if** (<logic_expr>) **then** <stmt> **else** <stmt> cümlesel biçimi için tek bir ayrıştırma ağacı vardır.

2.2. EBNF - Genişletilmiş BNF (Extenden BNF)

BNF'in bazı zorluklarısebebiyle üzerinde genişletme yapılmıştır. BNF'in çoğu genişletilmiş sürümü (hepsi de aynı olmamasına rağmen) EBNF olarak adlandırılır. Bu genişletmeler sadece okunabilirliği ve yazılabilirliği arttırmak için için yapılmıştır; BNF in tanımlayıcı gücünü arttırmak için değildir.

EBNF ile 3 yeni özellik gelmiştir. Bunlardan ilki, sağ el tarafındaki seçimlik kısmı tanımlamak için kullanılan [ ] işaretleridir. Seçimlik kısım köşeli parantezler içine yazılır. Örneğin C'de bir if-else ifadesi EBNF ile şöyle tanımlanabilir:

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

Eğer köşeli parantezler olmasaydı, sözdizimsel tanım iki kuralı ayrı ayrı belirterek yapılacaktı:

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

Eklenen diğer özellik yine sağ el tarafında kullanılan { } işaretleridir. { } içindeki ifade istenildiği kadar tekrar edilebilir veya hiç kullanılmayabilir.

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

Üçüncü eklenti çoktan seçmeli ifadeler için kullanılan | (VEYA) işaretinin parantez içinde kullanımıdır. Bir gruptan belli bir ifade seçileceğinde kullanılır.

Örn:
<term> → <term> (***** | **/** | **%**) <factor>

Yukarıdaki EBNF kuralı BNF ile ifade edilecek olursa üç ayrı kural yazılması gerekir:

<term> → <term> ***** <factor>
| <term> **/** <factor>
| <term> **%** <factor>

EBNF'te kullanılan [ ], { }, (|) gibi sembollere meta semboller denir; bunlar terminal semboller değillerdir. Eğer bu karakterler tanımlanan dilde terminal sembol olarak kullanılacaksa ya altı çizili ya da tırnak içinde belirtilmelidir.

Örn:
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>

<expr> → <expr> + <term> BNF kuralı, + işlemini soldan birleşimli olmaya zorlamaktadır. Fakat <expr> → <term> { (+ | -) <term> } EBNF kuralında birleşimin yönü belirli değildir. EBNF'i temel alan bir sözdizimi analizcisinde bu problem, sözdizimi analizi işleminin doğru birleşimi yapacak şekilde tasarlanması il ortadan kaldırılır. Bu konuya ilerleyen zamanlarda değineceğiz.

Bazı EBNF sürümlerinde } sağ kıvırcık parantezin üstünde yazılı sayılar bulunur. Bu sayılar ifadenin en çok kaç kere tekrar edebileceğini belirlemek için kullanılır. Bazı sürümlerde sayı yerine +

işareti kullanılır. + işareti ifadenin en az 1 kere tekrar edileceğini belirtir. Örneğin,

<compound> → begin <stmt> {<stmt>} end
ve
<compound> → begin {<stmt>}$^{+}$ end birbiri ile eşdeğer ifadelerdir.

3. Özellik Gramerleri

İçerikten bağımsız dillerin kısıtları

- Anlamsallığı gösteremezler.

Örn. Kullanılan değişken önce tanımlanmış olmalıdır veya değişkenin kullanıldığı yerlerde tip uyumu olmalıdır.

- Ayrıştırma ağacından başka birşey üretmekte kullanılamazlar.

Örn. Assembly kodu üretemeyiz

Özellik gramerleri

- İçerikten bağımsız gramerler ile tanımlanmış programlama dillerinin yapısının daha ayrıntılı tanımlanması için kullanılan bir araçtır. İçerikten bağımsız gramerlerin genişletilmiş şeklidir.
- Anlamsallık kontrolü ve diğer derleme zamanı analizleri için kullanılırlar.

Örn. Derleyicide tip uyumu kontrolü

- Çeviri için kullanılırlar.

Örn: Ayrıştırma ağacından assembly koduna çevirme

- Ayrıştırma ağacında dolaşmayı ve bu dolaşma sırasında bazı bilgilerin nasıl hesaplanacağını tanımlar.

3.1. Statik Anlamsallık

Programlama dillerinin bazı özelliklerini BNF ile tanımlamak zordur hatta bazen imkansızdır. Örneğin tip uyumluluğu kurallarını BNF ile tanımlamak zordur. Java'da bir kayan noktalı değer bir integer değere atanamaz (tersi mümkündür). Bu kısıtlama BNF ile tanımlanabilir ancak ek kurallar ve ek non-terminaller gerektirir. Eğer Java'daki tüm tip kuralları BNF ile tanımlanmış olsaydı, gramer çok büyük ve kullanışsız olurdu. Gramerin boyutu sözdizimi analizcisinin boyutunu belirler.

BNF ile tanımlanamayacak kurallara örnek olarak, değişkenlere referans verilmeden önce tanımlanmış olmaları gerektiği kuralını verebiliriz. Bu kuralın BNF ile tanımlanamayacağı ispat edilmiştir.

Bu tip problemler dilin statik anlamsal kurallarına örnektir. Bir dilin statik anlamsallığı, programın çalışma zamanındaki anlamı ile dolaylı yoldan ilişkilidir.Çoğu statik anlamsal kural, dilin tip kısıtlarından oluşmaktadır. “Statik anlamsallık” ismi, bu kuralların derleme zamanında kontrol edilebilmesinden dolayı kullanılmıştır.

BNF'in statik anlamsal kuralları tanımlamadaki yetersizliği nedeniyle, daha güçlü mekanizmalar tasarlanmıştır. Bu mekanizmalardan biri olan **özellik gramerleri,** programın hem sözdizimi hem de anlamsal kurallarını tanımlamak üzere tasarlanmıştır.

3.2. Temel Kavramlar

Özellik gramerleri içerikten bağımsız (context-free) gramerlerdir. İçerikten bağımsız gramerlere, **özellikler** (attributes), **özellik hesaplama fonksiyonları** (attribute computation functions) ve **karşılaştırma belirtimi fonksiyoları** (predicate functions) eklenmiş halidir.

Özellikler, gramer sembolleri (terminal ve non-terminaller gibi) ile birleştirilmiştir ve değer atanmış değişkenlere benzerler. **Özellik hesaplama fonksiyonları** (anlamsal fonksiyonlar da denir) gramer kuralları ile birleştirilmiştir. Özellik değerinin nasıl hesaplanacağını belirtmek için kullanılırlar. **Karşılaştırma belirtimi fonksiyonları**, gramer kurallarıyla ilişkili olan sabit anlamsal kuralları tanımlarlar.

3.3. Özellik Gramerleri Tanımı

- Her X gramer sembolü ile ilişkili bir A(X) özellik kümesi vardır. A(X) kümesi, S(X) ve I(X) ayrık kümelerinden oluşur. S(X) sentezlenmiş, I(X) de kalıtsal özellikler olarak adlandırılır. Sentezlenmiş özellikler ayrıştırma ağacında anlamsal bilginin yukarı taşınması, kalıtsal özellikler de anlamsal vernin aşağı taşınmasında kullanılır.
- Her gramer kuralı ile ilişkili olarak, gramer kuralındaki semboller üzerinde tanımlı anlamsal fonksiyonlar kümesi ve büyük ihtimalle boş olan bir karşılaştırma belirtimi fonksiyonları kümesi tanımlıdır. Bir $X_0 \rightarrow X_1,..,X_n$ kuralı için, X_0'ın sentezlenmiş özellikleri, $S(X_0)=f(A(X_1),...,A(X_n))$ formundaki anlamsal fonksiyonlarla hesaplanır. Böylece ayrıştırma ağacındaki bir düğümün sentezlenmiş özelliğinin değeri, o düğümün çocuk düğümlerinin özelliklerinin değerine bağlı olmaktadır.
  X_j, $(1 \leq j \leq n)$ sembollerinin kalıtsal özellikleri $I(X_j)=f(A(X_0),...,A(X_n))$ formundaki anlamsal fonksiyon ile hesaplanır. Böylece, ayrıştırma ağacındaki herbir düğümün kalıtsal özellikleri o düğümün ebeveyn veya kardeş düğümlerinden gelmektedir.
- Bir karşılaştırma belirtimi fonksiyonu, özellik birleşim kümesi üzerinde $\{A(X_0),...,A(X_n)\}$ mantıksal ifade şeklindedir. Bir özellik grameri ile izin verilen türetmeler, her non-terminalle

bağlantılı her karşılaştırma belirtimi **doğru** olduğunda yapılanlardır.

3.4. Yapı İçi Özellikler (Intrinsic Attributes)

Yaprak düğümlerin, değerleri ayrıştırma ağacı dışında kararlaştırılan sentezlenmiş özelliklerine yapı içi özellikler denir. Örneğin bir değişkenin tipi sembol tablosundan gelebilir. Sembol tablosu, değişkenlerin adlarının ve tiplerinin tutulduğu tablodur.

3.5. Özellik Gramerlerine Örnekler

Özellik gramerlerinin sabit anlamsallığı tanımlamada nasıl kullanılacağına dair basit bir örnek olarak ADA dilindeki alt yordamları verebiliriz. ADA'da bir alt yordam *end altyordam_adı* ile biter. Bu kural BNF ile gösterilemez.

Syntax rule: <proc_def> → procedure <proc_name> [1]
<proc_body> end <proc_name> [2];

Predicate: <proc_name> [1] string == <proc_name> [2] .string

Bir özellik gramerinde bir non-terminal birden fazla kullanılıyorsa birbirinden ayırmak için [] içinde numaralandırma yapılır. [] içindeki sayı veya parantezlerin kendisi dilin bir parçası değildir.

Yukarıda predicate kuralı ile gösterilen, alt yordamın başlığında bulunan <proc_name> non-terminalinin string özelliği alt yordamın end kelimesinden sonra gelen <proc_name> non terminalinin string özelliği ile aynı olmak zorundadır. Yani alt yordamın adı ile end'den sonra gelen adın eşleşmesi gerekir.

Şimdiki örnekte atama işlemlerinde tip kontrolü için özellik gramerlerinin nasıl kullanıldığını görelim. Atama işlemlerinin sözdizimi ve anlamsallığı şöyledir:

- Değişken isimleri sadece A, B veya C olabilir.
- Atamanın sağ tarafı bir değişken veya iki değişkenin toplandığı bir ifade olabilir.
- Değişkenlerin tipleri ya *int* ya da *real* olabilir.
- Eşitliğin sağ tarafında bulunan iki değişkenin tipleri aynı olmak zorunda değildir. Değişken tiplerinin farklı olduğu durumda ifadenin sonucu kesinlikle *real* olmalıdır.
- Eşitliğin sol tarafının tipi, sağ tarafın tipi ile uyumlu olmalıdır. Yani eşitliğin sağ tarafında karışık tipler bulunabiliyorken, sol tarafın tipi sağdaki ifadenin sonuç tipi ile aynı olduğu sürece eşitlik geçerli olur.

Özellik gramerleri bu sabit anlamsal özellikleri tanımlamaktadır.

Örnek özellik gramerimizin sözdizimi bölümü şöyledir:

<assign> → <var> = <expr>
<expr> → <var> + <var>
| <var>
<var> → A | B | C

Örneğimizdeki özellik gramerinin non-terminallerinin özellikleri ise şöyle tanımlanmıştır:

- *actual_type* (gerçek tip): <var> ve <expr> non-terminalleri ile ilişkili sentezlenen bir özelliktir. Int veya real şeklinde gerçek tipi tutmada kullanılır. Değişken durumunda yani <var> non-terminali için, gerçek tip yapı içidir (sembol tablosunda). <expr> non-terminalinin gerçek tipi ise çocuk düğümlerinin gerçek tiplerine göre kararlaştırılır.
- *expected_type* (beklenen tip): Kalıtsal bir özelliktir ve <expr> non-terminali ile ilişkilidir. İfadenin beklenen tipini tutar (buna atamanın sol tarafındaki değişkenin tipine bakarak karar verir).

Basit bir atama ifadesi için özellik grameri

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)

Burada look-up fonksiyonu adı verilen değişkenin sembol tablosundaki tipini döndürür. A=A+B için ayrıştırma ağacı: