BIMU4032
DERLEYİCİ TASARIMI

Bu Haftaki Konu Başlıkları

- **Belirsizlik (Ambiguity)**
- **Operatör Önceliği**
- **Belirsiz İfade Gramerlerini Yeniden Yazma**
- **Belirsizliğin Diğer Kaynakları**
- **Ayrıştırma (Parsing)**
- **Tahmin Edici Ayrıştırıcı**
- **Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)**

Belirsizlik (Ambiguity)

Eğer bir gramer benzer sırada benzer yapraklar ile farklı üretim ağacı üretebiliyorsa bu gramer belirsizdir. Bunun anlamı program metninin doğrusallığı nedeniyle üretim ağacını kaybettiğimiz zaman onun açıkça belirtilmiş anlamını kaybederiz ve anlam üretim ağacından çıktığı için anlam bilgisini de kaybederiz.

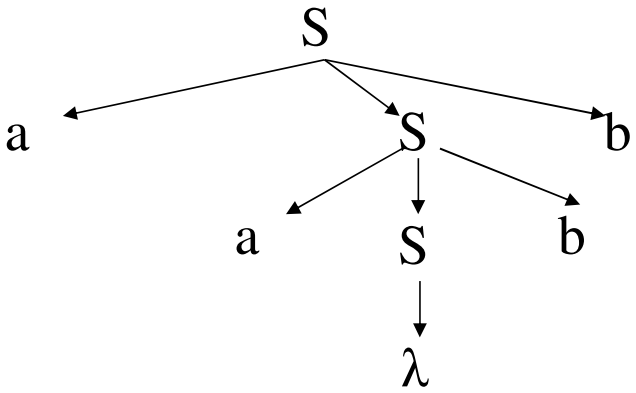
Aynı string için birden çok türetim yapılması bu string için derleyici tarafından birden çok kod kümesi oluşturulabilmesi anlamına gelir. Bu kod kümelerinden yalnız biri string'in beklenen amacına karşılık gelir. Diğer kod kümeleri ise, beklentiden farklı sonuçlar doğurur.

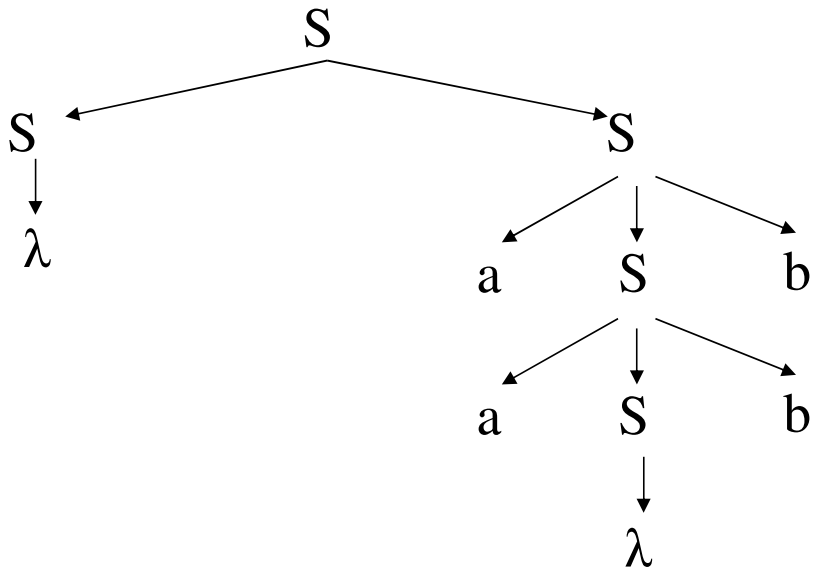
Belirsizlik (Ambiguity)

Bir ağacın oluşturulması sırasında türetimin sırası önemli olmamasına rağmen bir nonterminal için üretim seçeneği olabilir. Farklı seçenekler farklı string'lerin elde edilmesine neden olabilirler. Ne zaman bir gramer bazı string'ler için birkaç tane farklı syntax ağacı oluşturursa o zaman gramer belirsiz olarak adlandırılır. Gramer sadece string kümelerini tanımlamak için kullanılıyorsa belirsizlik bir sorun değildir. Ancak, gramer string'in yapısını yansıtmak için kullanılacaksa gramerin belirsiz olmaması istenir. Birçok durumda, belirsiz bir gramer aynı string'i üretecek belirsiz olmayan bir gramer olarak yazılabilir.

Belirsizlik (Ambiguity)

$S \rightarrow aSb \mid SS \mid \lambda$ türetim kurallarıyla verilen gramer, aabb dizgisi için belirsizdir.





Belirsizlik (Ambiguity)

Gramer-3'ün belirsiz şekli aşağıda verilmiştir.

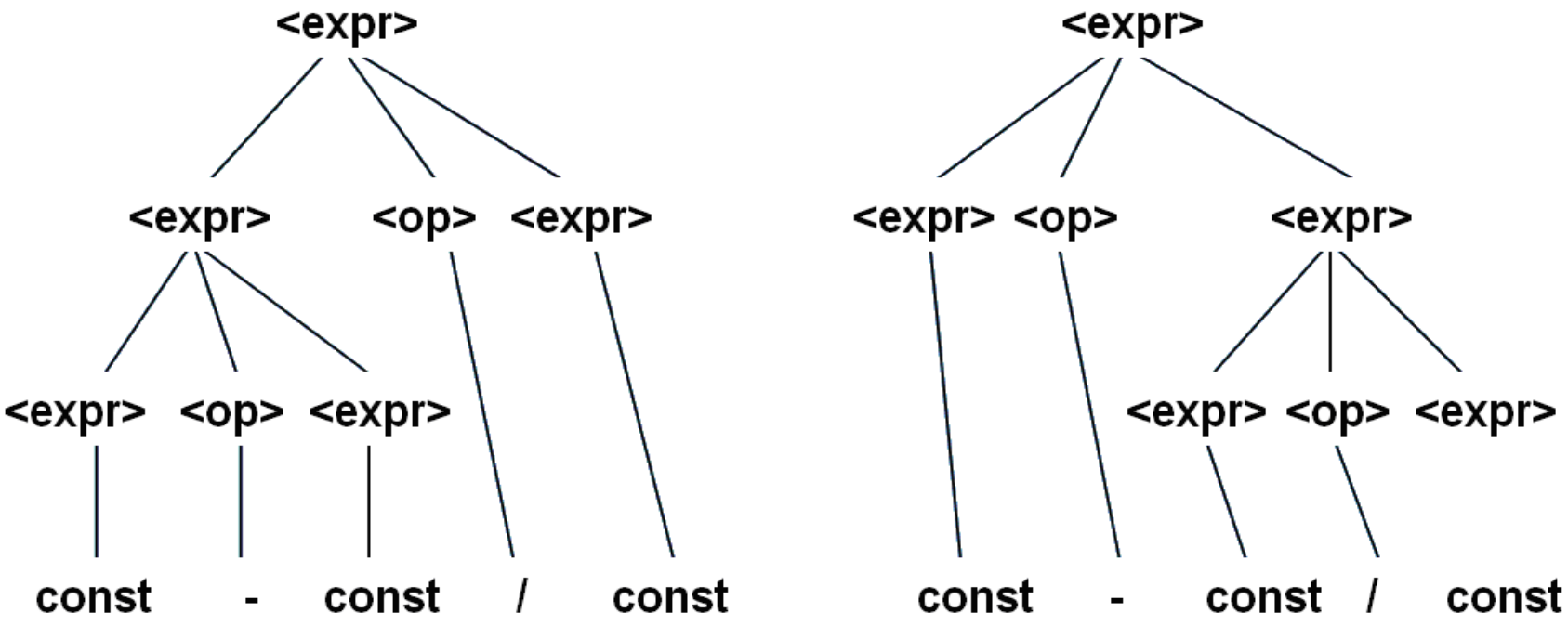
$$
\begin{aligned}
T &\rightarrow R \\
T &\rightarrow \mathtt{a}T\mathtt{c} \\
R &\rightarrow \\
R &\rightarrow \mathtt{b}R
\end{aligned}
$$

Ancak birçok durumda belirsizliğin tespit edilmesi ve kanıtlanması çok zordur. İyi bir parser'ın oluşturulması için belirsiz olmamanın kanıtlanması gereklidir.

Belirsizlik (Ambiguity)

expr → expr op expr | const

op → / | -



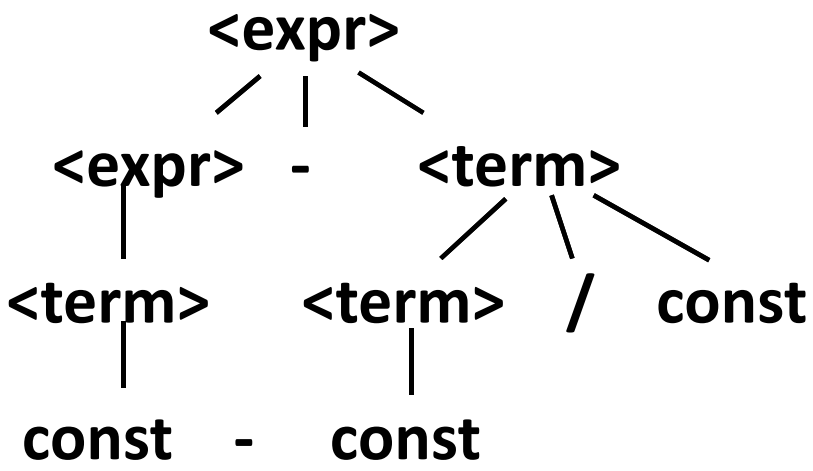
Belirsiz bir gramer örneği

Belirsizlik (Ambiguity)

Bu örnek için, eğer üretim ağacında işleçlerin(operators) öncelik seviyeleri gösterilirse belirsizlik olmaz.

<expr> → <expr> - <term> | <term>

<term> → <term> / const | const



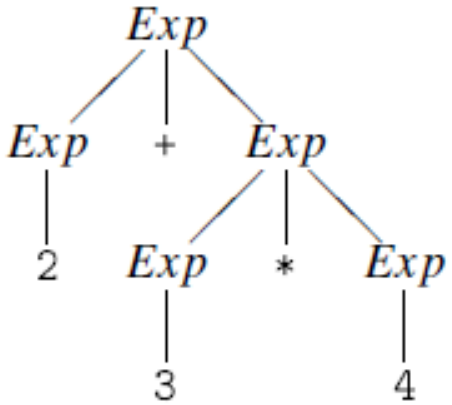
Operatör Önceliği

$$
\begin{aligned}
Exp &\rightarrow Exp + Exp \\
Exp &\rightarrow Exp - Exp \\
Exp &\rightarrow Exp * Exp \\
Exp &\rightarrow Exp \,/\, Exp \\
Exp &\rightarrow \mathbf{num} \\
Exp &\rightarrow (\, Exp \,)
\end{aligned}
$$

Gramerini inceleyecek olursak bu gramerin belirsiz bir gramer olduğu görülebilir. Bu belirsizlik 2+3*4 gibi bir ifade oluşturulduğunda bu ifadenin iki şekilde okunmasından kaynaklanmaktadır. İlk okumaya göre ifade 3 ile 4 çarpımının 2 ile toplanmasıyken ikinci okumaya göre 2 ile 3'ün toplamının 4 ile çarpımıdır. Basit elektronik hesap makineleri bu yorumlamalardan ikincisini seçerlerken daha karmaşık bilimsel hesap makineleri ve birçok programlama dili ilk yaklaşımı seçerler. Bu işlemi operatör önceliği hiyerarşisini kullanarak gerçekleştirirler.

Operatör Önceliği

Birçok programlama dili bilimsel hesap makineleri ile aynı düzeni kullanırlar. Bu nedenle gramerde bu açık hale getirilmelidir. İdeal olarak 2+3*4 ifade aşağıda gösterilen syntax ağacını üretir. Bu ağaç alt ifadelerin gruplanması yoluyla operatör önceliğini yansıtır. Bir ifade değerlendirilirken en üstteki operatör uygulanmadan önce syntax ağacının alt ağaçları tarafından simgelenen alt ifadeler değerlendirilir.



Operatör Önceliği

Belirsizliği çözmek için diğer bir yolda olası syntax ağaçları arasında seçim yapmak için syntax analiz süresince öncelik kurallarının kullanılmasıdır. Birçok parser bu yaklaşıma dayanır. Ancak bazı parserlar belirsiz olmayan grameri gerektirirler. Bu nedenle gramerin kendisinde operatör önceliğinin yansıtılması zorunludur. Bunu daha açık hale getirmek için bazı kavramlar aşağıda tanımlanmıştır.

- Eğer ifade a ⊕ b ⊕ c soldan sağa doğru değerlendiriliyorsa operatör ⊕ sol-birleşmelidir (a ⊕ b) ⊕ c.
- Eğer ifade a ⊕ b ⊕ c sağdan sola doğru değerlendiriliyorsa operatör ⊕ sağ-birleşmelidir a ⊕ (b ⊕ c).

Operatör Önceliği

- Genel kullanım olarak – ve / sol-birleşmelidir (2-3-4 ifadesi (2-3) - 4 şeklinde hesaplanır). + ve * matematiksel anlamda birleşmelidir. Bunun anlamı soldan sağa ya da sağdan sola hesaplama önemli değildir. Ancak belirsizlikten kaçınmak için bunlardan birisi seçilmelidir. Genel olarak sol-birleşmeli olarak seçilirler.

- Fonksiyonel dillerdeki liste yapıcı operatörler (:: ve @ gibi) sağ birleşmelidir. C dilindeki atama operatörü sağ birleşmelidir: a=b=c ifadesi a=(b=c) olarak okunur.

Belirsiz İfade Gramerlerini Yeniden Yazma

Eğer aşağıdaki gibi bir gramer varsa

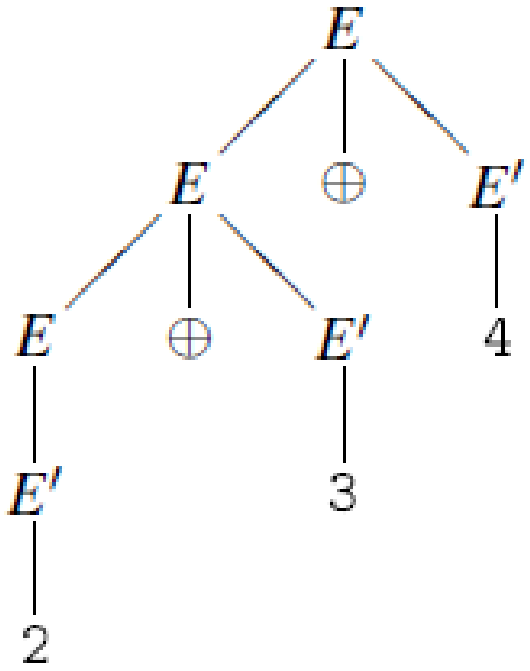
$$E \rightarrow E \oplus E$$
$$E \rightarrow \mathbf{num}$$

Bu gramer belirsiz olmayacak bir şekilde doğru yapıda yeniden yazılabilir. Bu $\oplus$ operatörünün birleşimine bağlı olarak farklı birleşimler için farklı yeniden yazma kuralları uygulanır. Eğer $\oplus$ operatörü sol-birleşmeliyse gramer operatör sembolünün sadece solu için rekürsif bir referans yoluyla sol-rekürsif hale getirilir:

$$E \rightarrow E \oplus E'$$
$$E \rightarrow E'$$
$$E' \rightarrow \mathbf{num}$$

Belirsiz İfade Gramerlerini Yeniden Yazma

Artık ifade $2 \oplus 3 \oplus 4$ sadece aşağıdaki şekilde ayrıştırılabilir.



Belirsiz İfade Gramerlerini Yeniden Yazma

Sağ-birleşmeli yöntemi de benzer şekilde gerçekleştirilir.

$$
\begin{aligned}
E &\rightarrow E' \oplus E \\
E &\rightarrow E' \\
E' &\rightarrow \mathbf{num}
\end{aligned}
$$

Birleşme özelliği olmayan operatörler için nonrekürsif üretimler de aşağıdaki şekilde gerçekleştirilir.

$$
\begin{aligned}
E &\rightarrow E' \oplus E' \\
E &\rightarrow E' \\
E' &\rightarrow \mathbf{num}
\end{aligned}
$$

Belirsiz İfade Gramerlerini Yeniden Yazma

Şu ana kadar yapılan uygulamalarda bir operatörün kendi cinsinden operatörler ile yaptığı işlemler incelendi. Bu durumu aynı önceliğe sahip birkaç operatör üzerine de uygulamak benzer şekildedir. Örneğin + ve – operatörleri için;

$$
\begin{aligned}
E &\rightarrow E + E' \\
E &\rightarrow E - E' \\
E &\rightarrow E' \\
E' &\rightarrow \mathbf{num}
\end{aligned}
$$

Belirsiz İfade Gramerlerini Yeniden Yazma

Aynı öncelikli operatörler bu çalışma için aynı birleşime sahip olmalıdırlar. Aynı nonterminal semboller için sol-birleşmeli ve sağ-birleşmeli üretimler grameri belirsiz yapar. Örneğin gramer;

$$
\begin{aligned}
E &\rightarrow E + E' \\
E &\rightarrow E' \oplus E \\
E &\rightarrow E' \\
E' &\rightarrow \mathbf{num}
\end{aligned}
$$

için uygulanan kural yeniden yazma kurallarına benzemektedir. + ve $\oplus$ operatörü aynı önceliğe ve farklı birleşimlere sahiptir. Fakat üretilen gramer yine belirsizdir ve aynı zamanda istenilen dili tanıyamaz. Örneğin **num+num** $\oplus$ **num** stringi bu gramer tarafından türetilemez.

Belirsiz İfade Gramerlerini Yeniden Yazma

Genelde, +'nın sol-birleşmeli ve $\oplus$ 'nin sağ-birleşmeli olduğu (ya da tam tersi) 1+2 $\oplus$ 3 gibi bir ifadede belirsizliği çözmek için belirgin bir yol yoktur. Bu nedenle birçok programlama dili aynı birleşim özelliğine sahip olmak için aynı öncelikte operatörler gerektirirler.

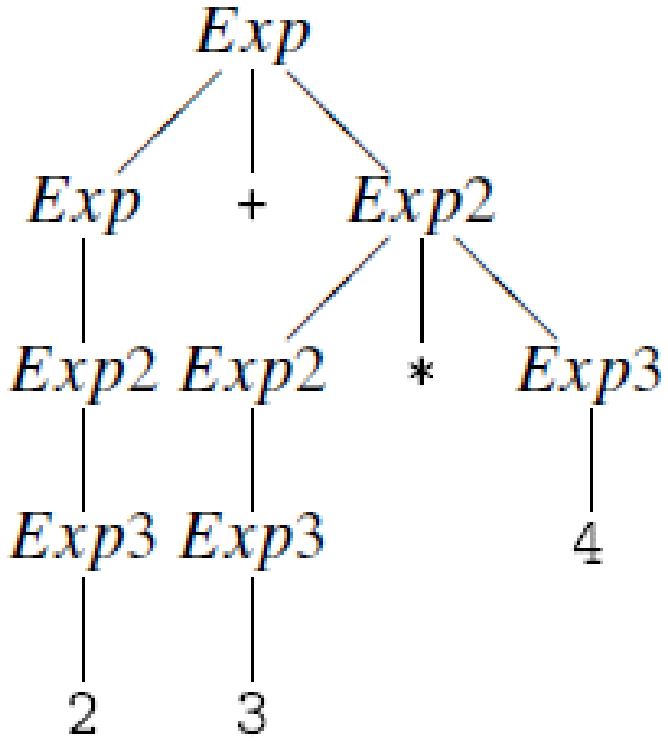
Aynı şekilde farklı öncelikteki operatörlerin de üstesinden gelinmelidir. Bu işlem her bir öncelik seviyesi için bir nonterminalin kullanılması ile gerçekleştirilir. Bu düşünce, ″eğer bir ifade belirli bir öncelik seviyesinin bir operatörünü kullanıyorsa onun alt ifadeleri daha düşük öncelikli operatörleri kullanamaz″ yaklaşımına dayanır. Bu nedenle bir öncelik seviyesine karşılık gelen bir nonterminal için üretimler sadece benzer önceliğe ya da daha yüksek önceliğe karşılık gelen nonterminaller için adlandırılır.

Belirsiz İfade Gramerlerini Yeniden Yazma

Gramer-1'i kullanan 2+3*4 ifadesinin syntax ağacı aşağıda verilmiştir.

$$
\begin{array}{lcl}
Exp & \rightarrow & Exp + Exp2 \\
Exp & \rightarrow & Exp - Exp2 \\
Exp & \rightarrow & Exp2 \\
Exp2 & \rightarrow & Exp2 * Exp3 \\
Exp2 & \rightarrow & Exp2 / Exp3 \\
Exp2 & \rightarrow & Exp3 \\
Exp3 & \rightarrow & \textbf{num} \\
Exp3 & \rightarrow & (\ Exp\ )
\end{array}
$$

Gramer-1 (belirsiz olmayan bir gramer)



Gramer-1 tarafından tanınabilen 2+3*4 ifadesinin syntax ağacı

Belirsizliğin Diğer Kaynakları

Programlama dillerindeki olası belirsizliklerin birçoğu syntax ifadelerinden gelir ve yeniden yazma kuralları kullanılarak düzeltilebilir. Diğer bir klasik belirsizlik örneği de “dangling else” problemidir.

Pascal ve C gibi programlama dilleri *else* kısmının kullanılma durumunu seçimli yaparlar. Bu nasıl ayrıştırma yapılacağının açık olmaması sorunudur.

Belirsizliğin Diğer Kaynakları

- Consider the Grammar for if-then-else statements:

$$
\begin{aligned}
\mathit{Stmt} \;&\rightarrow\; \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt} \\
&\mid\; \textbf{if}\ \mathit{Expr}\ \textbf{then}\ \mathit{Stmt}\ \textbf{else}\ \mathit{Stmt} \\
&\mid\; \textbf{other}
\end{aligned}
$$

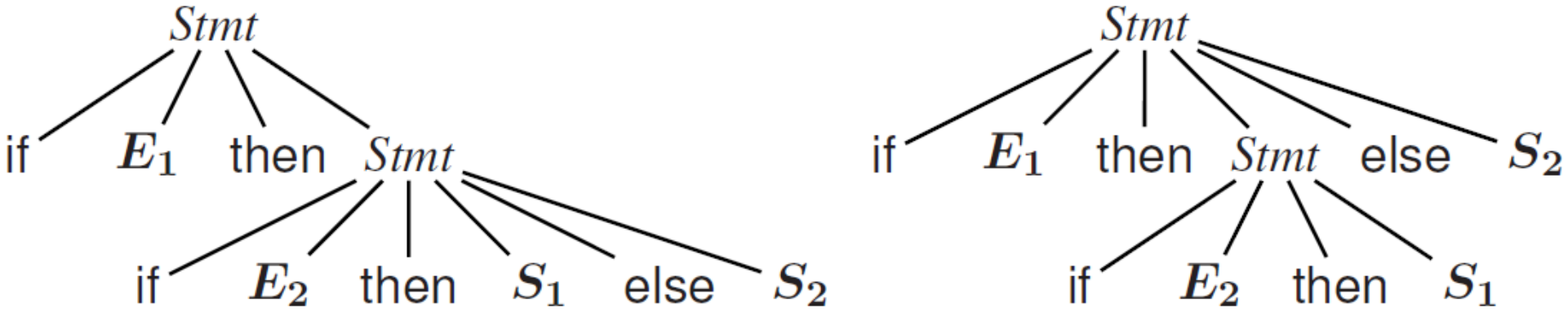
- This Grammar is ambiguous.
- **Example.** Consider the statement:

$$\text{if } E_1 \text{ then if } E_2 \text{ then } S_1 \text{ else } S_2.$$

Gramerine göre, *else* komutu eşit sayıda her bir *if* komutu ile eşleşebilir. Genel kullanıma göre bir *else* komutu daha önce eşleştirilmeyen en yakın *if* komutu ile eşleşebilir. Bu örneğe göre else ilk *if* ile mi yoksa ikinci *if* ile mi eşleşir? Bu belirsizlik gramer içinde açık hale getirilmelidir.
If, then ve *else* komutları sağ-birleşmeli operatörlerin bir çeşidi olarak işlendiğinde bir *if-then* bloğunu en yakın *else* ile eşleştirir. Ancak burada önceki yeniden yazma kuralları yazılamaz.

Belirsizliğin Diğer Kaynakları

The statement: if E_1 then if E_2 then S_1 else S_2, has two Parse-Trees



Typically, the first Parse-Tree is preferred.

Disambiguating Rule: Match each else with the closest unmatched then.

Belirsizliğin Diğer Kaynakları

The rule can be incorporated into the Grammar if we distinguish between *matched* and *unmatched* statements.

A statement between a then-else must be *matched*.

```
Stmt           → Matched_stmt | Unmatched_stmt
Matched_stmt   → if Expr then Matched_stmt else Matched_stmt
               | Other-Stmt
Unmatched_stmt → if Expr then Stmt
               | if Expr then Matched_stmt else Unmatched_stmt
```

This Grammar generates the same set of strings as the previous one but gives just one Parse-Tree for if-then-else statements.

Ayrıştırma (Parsing)

Bir derleyicinin syntax analiz fazı, lexer tarafından üretilen tokenlerin bir stringini alacak ve bu string için gramerin başlangıç sembolünden stringin türetimini bularak bir syntax ağacı oluşturacaktır. Bu işlem ayrıştırma (parsing) olarak da adlandırılır.

Ayrıştırma işleminin amacı, verilen bir stringin **(w)**, belirli bir gramer **(G)** tarafından türetilip türetilmediğinin bulunması, eğer string gramer tarafından türetilmişse de nasıl türetildiğine ilişkin bilginin elde edilmesidir. Eğer ayrıştırma ağacı elde edilemiyorsa, verilen stringin **(w)** ilgili gramer **(G)** tarafından türetilmediği; başka bir deyişle sözdizimi açısından **w**'nin yanlış olduğu sonucu çıkarılır. Eğer ayrıştırma ağacı elde edilir, başka bir deyişle **w**'nin elde edilebilmesi için **G**'nin hangi kurallarının hangi sırada uygulanması gerektiği ortaya konulursa, hem **w**'nin sözdizimi açısından doğru olduğu sonucu çıkarılır, hem de **w**'nin taşıdığı **anlam (semantic)** belirlenmiş olur.

Ayrıştırma (Parsing)

Bu işlem doğru bir tane bulana kadar tahmini türetimler yapılarak devam eder. Fakat rastgele tahminler oldukça etkili bir yöntemdir. Hatta bazı ayrıştırma yöntemleri “tahmin” türetimlerini temel alır. Bu yöntem tahmin edici (predictive) ayrıştırma yöntemi olarak adlandırılır. Tahmin edici ayrıştırıcılar syntax ağacını daima kökten yapraklara doğru oluşturduğu için aynı zamanda yukarıdan-aşağıya (top-down) ayrıştırıcılar olarak da adlandırılırlar.

Ayrıştırma (Parsing)

Diğer ayrıştırıcılar farklı bir yolu kullanırlar: üretimlerin sağ tarafları ile uyumlu giriş string'inin parçalarını ararlar ve bunu sol taraf nonterminalleri için tekrar yazarlar (aynı anda syntax ağacının parçalarını oluştururlar). Başlangıç sembolü ile string tekrar yazıldığı zaman sonuç olarak syntax ağacı da tamamlanmış olur. Bu tür yöntemler aşağıdan-yukarıya (bottom-up) ayrıştırıcılar olarak adlandırılırlar.

Yukarıdan aşağıya ayrıştırma yazılabilir ya da otomatik (aşağı-itme otamatı) olarak oluşturulabilir. Ancak aşağıdan yukarı ayrıştırma sadece otomatik olarak üretilebilir.

Tahmin Edici Ayrıştırıcı

$T \rightarrow R$

$T \rightarrow \mathtt{a}T\mathtt{c}$

$R \rightarrow$

$R \rightarrow R\mathtt{b}R$

$\underline{T}$

$\Rightarrow \mathtt{a}\underline{T}\mathtt{c}$

$\Rightarrow \mathtt{aa}\underline{T}\mathtt{cc}$

$\Rightarrow \mathtt{aa}\underline{R}\mathtt{cc}$

$\Rightarrow \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{cc}$

$\Rightarrow \mathtt{aa}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc}$

$\Rightarrow \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{cc}$

$\Rightarrow \mathtt{aab}\underline{R}\mathtt{b}R\mathtt{b}R\mathtt{cc}$

$\Rightarrow \mathtt{aabb}\underline{R}\mathtt{b}R\mathtt{cc}$

$\Rightarrow \mathtt{aabbb}\underline{R}\mathtt{cc}$

$\Rightarrow \mathtt{aabbbcc}$

Buradaki soldan-türetime bakıldığında tekrar yazılan nonterminallerin sol tarafında sadece terminal sembollerin kaldığı görülür. Bu terminal semboller ayrıştırılan string'in öneki için uygundur. Bir ayrıştırma durumunda bu önek çoktan okunmuş girişin bir parçası olacaktır. Parser'ın görevi şimdi en sol nonterminalin yeniden yazılacağı zaman hangi üretimin kullanılacağını seçmektir.

Gramer-3

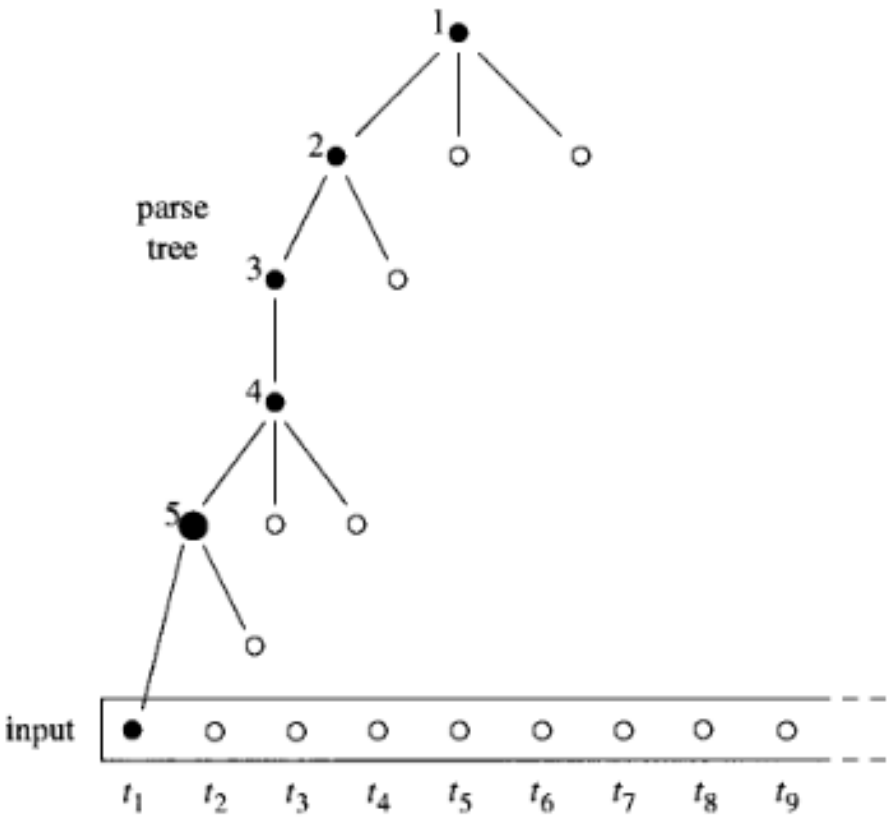
Tahmin Edici Ayrıştırıcı

Eğer şekildeki üçüncü satıra bakarsak önceden iki a'nın okunduğunu ve bir sonra gelecek sembolün b olduğu görülür. $T \longrightarrow aTc$ üretiminin sağ tarafı bir tane a ile başladığı için bu üretimin kullanılamayacağı açıktır. Burada T'nin $T \longrightarrow R$ üretimi seçilir.

Bir sonraki adımda ise R için türetimlerin hiç birisi bir terminal sembol ile başlamıyor. Bu nedenle hemen bir türetim seçilemez. Bu gramerin belirsiz bir gramer olduğu düşünülürse daima tek bir seçimin yapılamaması doğaldır. Eğer bunun yerine belirsiz olmayan bir gramer olsaydı R'nin ikinci türetimi hemen seçilebilirdi. Tüm b'ler okunduğu zaman bir sonraki c'de olunacaktır. Burada R için boş üretimler seçilir ve türetilen stringin geri kalanı ile üretilen string eşleştirilir.

Eğer bir sonraki giriş sembolüne dayalı daima tek bir üretim kuralı seçilebiliyorsa tahmin edici ayrıştırıcı uygulanabilir.

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)



A top-down parser recognizing the first token in the input.

Ağacın kökünün oluşturulması ile başlar. Sözdizimi ağacının düğümleri önce kök ile oluşturulur. Yukarıdan aşağıya ayrıştırma bir düğümü oluşturduğu zaman düğümün etiketi kendisi tarafından önceden biliniyordur. Girişteki bilgiyi kullanarak N için doğru seçenekleri belirler. Hangi seçeneğin uygulanacağının bilinmesi N ile etiketlenmiş bu düğümün bütün çocuk düğümlerinin etiketlerinin bilinmesini sağlar. Daha sonra ayrıştırıcı N'nin ilk çocuk düğümünü oluşturmaya devam eder. Bu süreç en sol düğümün terminal sembol olmasına kadar devam eder. Terminal sembol programda ilk token (t1) ile uyumludur.

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)

Yukarıdan-aşağıya ayrıştırmada, gramerin başlangıç simgesi olan **S**'den başlanarak, ayrıştırılacak string **(w)** türetilmeye çalışılır. **w** türetilmeye çalışılırken soldan türetme kuralları uygulanır. Soldan türetmede, türetmenin herhangi bir adımındaki stringsel yapı **α** ise, ve **α** içindeki sözdizim değişkenlerinden en soldaki A ise, A kurallarından biri uygulanarak yeni bir string yapı elde edilir:

$S \rightarrow \alpha, \qquad \alpha = uA\beta \; : \alpha \in V^{+} \qquad \beta \in V^{*} \qquad A \in V_N \qquad u \in V_{T^*}$

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)

Salt uç simgelerden oluşan **u,** string yapının terminal önekidir. **u**'nun aynı zamanda ayrıştırılan sözcüğün **(w)** de öneki olması gerekir. Eğer **u w**'nin öneki değilse, **α = uAβ** string yapısından **w** türetilemez. Yukarıdan-aşağıya ayrıştırma süreci bir ağaçla gösterilirse, bu ağaç aşağıdaki gibi oluşturulur:

- Ağacın kökünün etiketi **S**'dir.
- Gramerin her ($S \Longrightarrow \alpha$) yeniden yazma kuralı için, ağacın 2. düzeyinde etiketi α olan bir düğüm oluşturulur.
- Ağacın herhangi bir düğümünün etiketi **uAβ** ise, bu düğümün altında, her **(A**$\Longrightarrow$ϒ) yeniden yazma kuralı için, etiketi **u**ϒ**β** olan yeni bir düğüm oluşturulur.

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)

- Ağacın herhangi bir düğümünün etiketi **uAβ** ise, ve de **u** **w**'nin öneki değilse, bu düğüm bir ölü düğümdür ve bu düğümün altında türetme yapılamaz. Eğer bir düğümün etiketi salt **u** ise, ve de **u** **w**'den farklı ise bu düğüm de bir ölü düğümdür.
- Türetme işlemi ağacın bir düğümünün etiketi **w** oluncaya, ya da ağacın tüm yaprakları birer ölü düğüm oluncaya kadar sürdürülür. Eğer etiketi **w** olan bir düğüm elde edilirse ayrıştırma olumlu sonuçlanmış olur. Eğer etiketi **w** olan bir düğüm elde edilemez ve ağacın tüm yapraklarının birer ölü düğüm olduğu anlaşılırsa, ayrıştırma olumsuz sonuçlanmış olur.
- Eğer gramer belirsiz olmayan bir gramer ise, ağacın birden çok düğümünün etiketi aynı olamaz.

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)

Yukarıdan-aşağıya ayrıştırma işlemi deterministik olmayan bir işlemdir. Ayrıştırma işleminin karmaşıklığı, gramerin yeniden yazma kurallarının sayısı ile ayrıştırılan string'deki simge sayısına paralel olarak üssel biçimde artar. Yukarıdan-aşağıya ayrıştırma için kullanılabilecek iki algoritma vardır. Bu algoritmalar:

- Genişlik-öncelikli yukarıdan-aşağıya ayrıştırma (Breadth-first top-down parsing)
- Derinlik-öncelikli yukarıdan-aşağıya ayrıştırma (Depth-first top-down parsing)

algoritmaları olarak adlandırılır. Her iki algoritma da deterministik değildir. Oysa uygulama açısından ayrıştırma işleminin deterministik olması son derece önemlidir.

Yukarıdan-Aşağıya Ayrıştırma (Top-down Parsing)

Örnek: w=aababbaba string'inin ayrıştırılması

S → aS | AB | B

A → abA | ab

B → BB | ba