Sözdizimi ve Anlam (Syntax and Semantics) Tanımlama

Programlama dillerinin de doğal diller gibi sözdizimi ve anlam bakımından kurallara uygunluk testini geçmesi gerekmektedir. **Sözdizimi** (syntax), bir dilin ifadelerini, yapılarını ve program birimlerini belirtir. Bu bileşenlerin ne anlama geldiği de dilin **anlamsal** (semantic) yapısını oluşturur.

Örneğin Java'da while yapısının sözdizimi şöyledir:

while (boolean_expr) statement

Bu ifadenin anlamı ise şöyledir: boolean_expr'ın o anki değeri true ise statement çalıştırılır, değilse kontrol while'dan sonraki yapıya geçer. Döngünün tekrarı için ise kontrol, boolean_expression'a kapalı bir şekilde döner.

Sözdizimi ve anlam birbiriyle yakından ilgilidir. İyi tasarlanmış dillerde anlamın ne olduğu sözdiziminden çıkarılabilmelidir. Sözdizimi tarifi anlam tarifinden daha kolaydır, çünkü evrensel olarak kabul edilen bir gösterim biçimi vardır.

1. Sözdizimi tanımlama

Doğal (örneğin Türkçe) ya da yapay (örneğin Java) tüm diller, bir alfabeye ait karakterler dizisidir. Dillerde kullanılan karakter dizilerine **cümle** (sentence) veya **ifade** (statement) denir. Hangi karakter dizilerinin bir dilde geçerli olduğunu gösteren kurallara **sözdizim kuralları** (syntax rules) diyoruz.

Bir dile ait veri sözlüğünün temel birimine **lexeme** denir. Lexeme'lerin tanımı sözdizimsel tanımdan bağımsız olarak, lexical belirtimlerle yapılır. Lexemeler sayılar, işleçler, özel kelimeler gibi şeyler olabilir. Lexemeler gruplara ayrılırlar; örneğin değişken, metod, sınıf isimleri birer tanımlayıcıdır (identifier). Her lexeme grubu bir isim veya token ile temsil edilir. Bir **token**, lexemelerin kategorisi olarak tanımlanır.

Örn: index = 2 * count +17 ;

Lexeme'ler	Token'lar
index	identifier
=	equal_sign
2	int_literal
*	mult_op
count	identifier
+	plus_op
17	int_literal
;	semicolon

1.1. Dil Tanıyıcıları (Recognizers)

Diller iki şekilde tanımlanır: Tanımlama ile, üretimle. **L** dilinin **Σ** alfabesini kullanan bir dil olduğunu varsayalım. **Σ** alfabesindeki karakterlerden oluşan bir

dizinin **L** diline ait olup olmadığına karar vveren bir **R** tanıyıcısına ihtiyaç vardır. Derleyicideki sözdizimi analizcisi, derleyicinin dönüştürdüğü dil için bir tanıyıcıdır.

1.2. Dil üreticileri (Generators)

Bir dil üreticisi dile ait cümleler oluşturan bir araçtır. Yazılan bir ifadenin sözdizimi doğruluğunu kontrol ederken dil üreticisinin yapısı ile karşılaştırma yapılır. Bir dil için üretici ve tanıyıcı arasında yakın ilişki vardır.

2. Sözdizimi Tanımlamada Kullanılan Resmi Yöntemler

Bu bölümde gramer adı verilen ve bir programlama dilinin sözdizimini tanımlamaya yarayan resmi dil üretim mekanizmalarından bahsedilecektir.

2.1. Backus- Naur Form ve İçerikten Bağımsız Gramerler

2.1.1. İçerikten Bağımsız Gramerler (Context-Free Grammars)

1950'lerin ortasında bir dilbilimci olan Noam Chomsky dört adet dil sınıfı tanımlamıştır. Bunlardan, **içerikten-bağımsız** ve **düzenli** olarak adlandırılan ikisi, programlama dillerinin sözdizimi tanımlaması için de uygundur. Bütün programlama dilleri, küçük istisnalar hariç, içerikten-bağımsız gramerlerle tanımlanabilmektedir.

2.1.2. Backus – Naur Form (BNF)

BNF, Programlama dillerinin sözdizimi tasarımında kullanılan resmi bir notasyondur. Chomsky'nin tanımlamalarının üzerine, 1959'da John Backus tarafından, bir programlama dilinin sözdizimi tanımı için resmi bir notasyon yayınlanmıştır. Kısa bir süre sonra, 1960'ta Peter Naur tarafından üzerinde küçük değişiklikler yapılarak ALGOL60'ın sözdizimi tasarımında kullanılmıştır. Bu değişikliklerden sonra bu resmi notasyon BNF(Backus-Naur Form) adını alarak, programlama dillerinin sözdizimi tanımlaması için kullanılmaya başlamıştır.

Bir metadil, başka bir dili tanımlamak için kullanılan bir dildir. Programlama dilleri için de metadil BNF'tir. BNF, cümlesel yapılar için soyutlamalar kullanır. Örneğin Java’da basit bir atama ifadesi BNF ile şu şekilde gösterilebilir:

<assign> → <var> = <expression>

Okun sol tarafındaki soyutlamaya sol el tarafı denir (left hand side, LHS). Okun sağ tarafında bulunanlar, sol el tarafının tanımıdır. Okun sağ tarafı, sağ el tarafı (right hand side, RHS) olarak adlandırılır ve lexemeler, tokenlar ve başka soyutlamalara yapılan referanslardan oluşur. Tanımın tümüne kural (rule) veya üretim (production) denir.

Bir BNF tanımında veya gramerde bulunan soyutlamalara non-terminal semboller veya kısaca non-terminaller denir. Lexemeler ve tokenlar ise terminal semboller veya terminaller olarak adlandırılır.

BNF tanımı veya gramer, bir kurallar bütünüdür. Non-terminaller, iki ya da daha

fazla farklı tanıma sahip olabilirler. Çoklu tanımlar aralarına | (VEYA) işareti konularak tek bir kural halinde yazılabilirler.

Java'da if yapısı BNF ile şu şekilde gösterilebilir:

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

VEYA

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

Burada <stmt> tek bir ifade olabileceği gibi, birleşik ifadeleri de temsil edebilir. Basit olmasına rağmen BNF, programlama dillerinin sözdiziminin neredeyse tamamını tanımlamak için oldukça yeterlidir.

2.1.3. Liste Tanımlama

Değişken uzunluktaki listeler matematikte (...) ile gösterilir. Örn: 1,2,3, ... BNF'te bu gösterim yoktur, dolayısıyla alternatif bir yönteme ihtiyaç vardır: özyineleme.

<ident_list> → identifier | identifier, <ident_list>

Eğer kuralın solundaki bir non-terminal okun sağ tarafında da görülüyorsa bu kural özyinelemelidir.

2.1.4. Gramerler ve Türetmeler(Derivations)

Gramer, dil tanımlama için üretici bir araçtır. Dilin kurallarını sırasıyla uygulayarak dile ait cümleler oluşturulur. Cümle oluşturmaya **başlangıç sembolü** (start symbol) denen özel bir non-terminalden başlanır. Bir gramere ait kuralların sırayla uygulanması ile bir cümle üretmeye **türetme** (derivation) diyoruz. Örneğin bütün bir dili tanımlıyorsak, başlangıç sembolü genellikle tüm programı temsil eden <program> non-terminalidir.

Örn. 1:

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

- Yukarıda tanımlanmış basit gramerde sadece bir deyim var o da atama.
- Program begin kelimesi ile başlıyor, noktalı virgülle birbirinden ayrılan deyimler listesi ile devam ediyor ve end kelimesiyle bitiyor.
- Bir deyim bir değişken veya + ve - ile ayrılmış iki değişken olabilir.
- Değişken isimleri sadece A, B veya C olabilir.

Bu dilde aşağıdaki gibi bir türetme yapabiliriz.

<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

Bu türetme, tüm diğer türetmeler gibi bir başlangıç sembolü ile başlar ve buradaki örnekte bu sembol <program> dır. => sembolü, “türetir” olarak okunur.
Sıradaki her ardışık metin, bir önceki metindeki bir non-terminali, tanımlarından biriyle değiştirerek yapılmaktadır. Türetmedeki her metin, <program> da dahil olmak üzere, **cümlesel biçim** (sentential form) olarak adlandırılır. Bu türetmede değiştirilen non-terminal bir önceki cümlede en solda bulunan non-terminaldir. Bu tipteki türetmelere **en sol türetmeler** (leftmost derivations) denir. Türetme işlemi, cümlesel formda hiçbir non-terminal kalmayıncaya kadar devam eder. Türetmenin hangi yönden yapıldığının gramerin ürettiği dile bir etkisi yoktur. Türetmelerde non-terminallerin yerine yazılan tanımları değiştikçe, o dile ait farklı cümleler elde edilebilir.

Örn. 2:

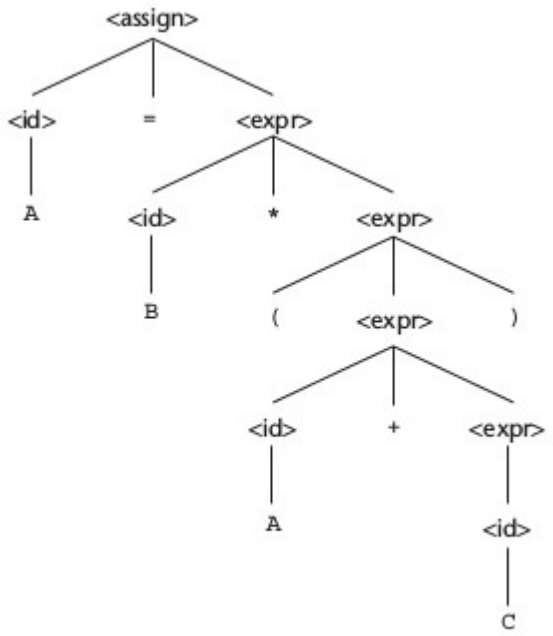
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

Bu gramer ile A = B * (A + C) cümlesi şu şekilde türetilebilir:

<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)

2.1.5. Ayrıştırma Ağaçları (Parse Trees)

Gramerler, tanımladıkları dilin cümlelerinin hiyerarşik sözdizim yapısını doğal yoldan tanımlayabilirler. Bu hiyerarşik yapılara ayrıştırma ağacı diyoruz. **Örn. 2**'deki türetme için ayrıştırma ağacı **Şekil 1**'deki gibidir.



Şekil 1. A = B * (A + C) ifadesi için ayrıştırma ağacı

Bir ayrıştırma ağacının her iç düğümü bir non-terminaldir. Ağacın yapraklarını ise terminaller oluşturur.

2.1.6 Belirsizlik (Ambiguity)

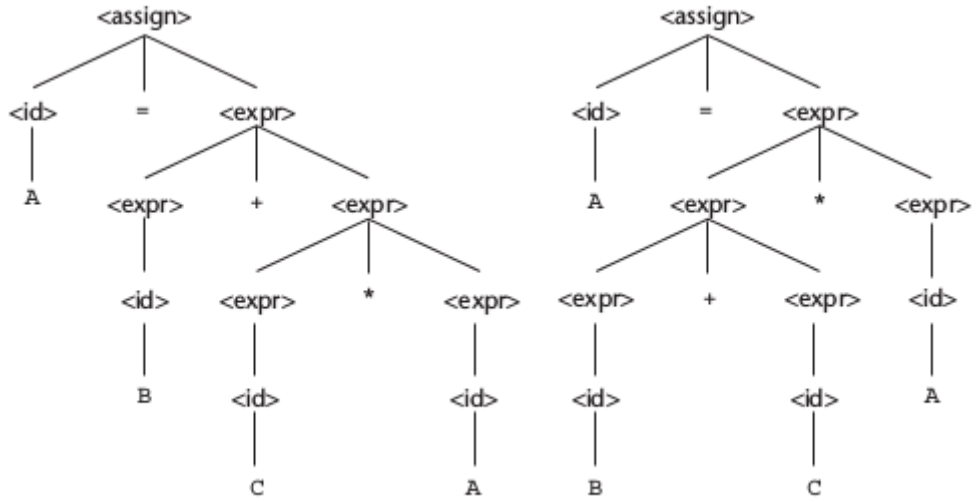
Bir cümlesel biçim için iki ya da daha fazla ayrıştırma ağacı üreten gramere, belirsiz (ambigious) gramer denir.

Örn.3:
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr> | <expr> * <expr> | (<expr>) | <id>

Yukarıda belirsiz bir gramer gösterilmiştir. Bu gramer belirsizdir, çünkü bu gramerle türetilmiş A = B + C * A cümlesi iki ayrı ayrıştırma ağacına sahiptir. **Örn.2** 'deki dile göre daha az cümlesel yapı tanımlandığı için belirsizlik oluşmaktadır. Burada ayrıştırma ağacı sadece sağa doğru değil, hem sağa hem de sola doğru büyüme göstermektedir(**Şekil 2**).

Dilin yapısında sözdizimsel belirsizlik (syntactic ambiguity) olması bir problemdir. Çünkü derleyiciler bu sözdizimsel biçimleri temel alarak hazırlanır. Derleyici kodun ayrıştırma ağacını inceleyerek üretilecek kodu belirler. Birden fazla ağaç olması durumunda o yapının anlamı tekil olarak kararlaştırılamaz.
Belirsizlik olup olmadığını anlayabilmek için diğer yollar:
- Bir cümle birden fazla en sol türetme ile türetilebiliyorsa

- Bir cümle birden fazla en sağ türetme ile türetilebiliyorsa, o gramer belirsizdir.



Şekil 2. A = B + C * A ifadesi için oluşan iki farklı ayrıştırma ağacı

2.1.7. İşleç önceliği (operator precedence)

Bir ifade iki tane farklı işleç içerdiğinde (örn. x+y*z), bu işleçlerden önce hangisinin işlem göreceği anlamsal bir problemdir. Bu sorun işleçlere farklı öncelik düzeyleri atanarak ortadan kaldırılabilir.

Bir ifadenin anlamının bir kısmı, o yapıya ait ayrıştırma ağacından çıkarılabilir. Önceliği yüksek olan işleç, ayrıştırma ağacında daha aşağı seviyede olmalıdır.

Şekil 2'deki ağaçları ele alalım. İlk ağaçta çarpma işlemi öncelikliyken, ikinci ağaçta toplama işlemi çarpmadan daha aşağı seviyede yer almaktadır. Bu iki ağaç, işleç önceliği bakımından birbiri ile çelişen sonuçlar vermektedir.

Örn. 2'de, gramer belirli olmasına rağmen işleç önceliği alışılmışın dışındadır. Bu gramerde çoklu işlecin bulunduğu bir cümlenin en sağdaki işleci, ayrıştırma ağacında en alt seviyede yer alır.

Örneğin A+B*C de *
A*B+C de + işleçleri en alt seviyede yer alır ve öncelikli işleçtirler.

İşleç önceliği probleminin çözümü için yapılması gereken, her işleç için yeni bir non-terminal ifadeyi ve yeni kuralları gramere eklemektir.

Örn. 4:
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> * <factor> | <factor>
<factor> → (<expr>) | <id>

+ ve * işleçlerinin ikisi için de <expr> kullanmak yerine, üç tane non-terminal kullanabiliriz. Bu kullanım işleçleri ayrıştırma ağacında farklı seviyelere yerleşmeye zorlar. Eğer ifadeler için kök sembol <expr> ise, <expr> non terminaline + işlemi ürettirerek ve yeni kullandığımız <term> non-terminalini + işleminin sağ tarafına yazarak + işlecini ağacın üst seviyelerinde olmaya zorlayabiliriz. Daha sonra <term> non-terminalini * işlecini üretmek üzere tanımlayabiliriz. Bu tanımda <term> sol tarafta ve bir diğer yeni non-terminal olan <factor> de sağ tarafta kullanılır. Bu yeni tanımlama ile * işleci, + işlecinden önce işlem görür. Çünkü her türetmede, * işareti başlangıç sembolüne + işaretinden daha uzakta olacaktır.

Örn. 4'teki yeni gramerle, daha önceki gramerlerle de üretebildiğimiz aynı dili üretebiliriz ancak üreteceğimiz dil hem belirli (unambigious) olur hem de işleç önceliği doğru tanımlanmış olur. Yeni gramerle **A=B+C*A** cümlesini en soldan ve en sağdan türetelim.

En soldan türetme:

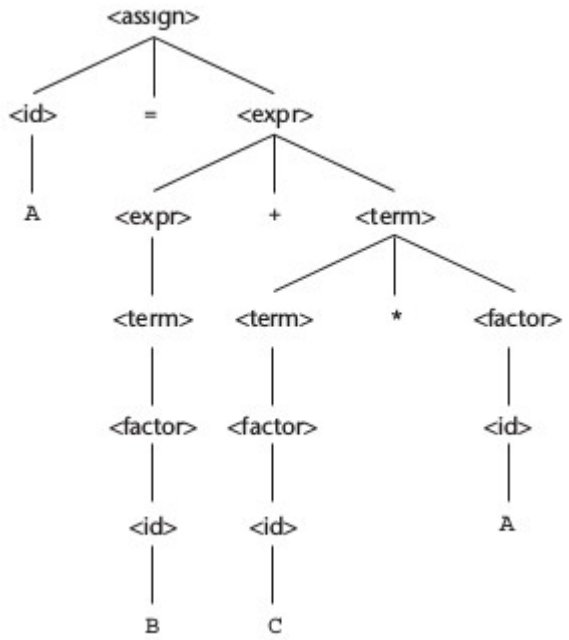
<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <term>
=> A = <term> + <term>
=> A = <factor> + <term>
=> A = <id> + <term>
=> A = B + <term>
=> A = B + <term> * <factor>
=> A = B + <factor> * <factor>
=> A = B + <id> * <factor>
=> A = B + C * <factor>
=> A = B + C * <id>
=> A = B + C * A

En sağdan türetme:

<assign> => <id> = <expr>
=> <id> = <expr> + <term>
=> <id> = <expr> + <term> * <factor>
=> <id> = <expr> + <term> * <id>
=> <id> = <expr> + <term> * A
=> <id> = <expr> + <factor> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <term> + C * A
=> <id> = <factor> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

İster en sağdan, ister en soldan türetme yapalım, aynı cümle için ayrıştırma ağacı aynı olacaktır. Çünkü **belirli** (unambigious) bir gramerle türetilmiş bir cümle için, türetme yönteminden bağımsız olarak, aynı ayrıştırma ağacı elde

edilir (**Şekil 3**). Ayrıştırma ağaçları ve türetmeler birbiri ile yakından ilişkilidir ve birinden diğerine kolaylıkla dönüşüm yapılabilmektedir.



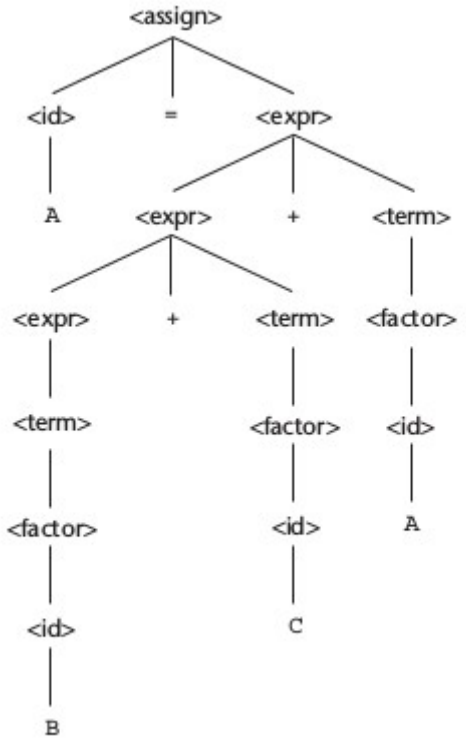
Şekil 3. Örn. 4 teki gramerle elde edilen A= B+C*A cümlesi için ayrıştırma ağacı

2.1.8. İşleçlerin birleşikliği (Associativity of Operators)

Bir ifadede aynı önceliğe sahip işleçler varsa (örn: A/B*C gibi) hangi işlecin daha önce çalıştırılacağına karar vermek mantıksal bir kurala bağlanır ve bu kurala **birleşiklik** (associativity) denir. Aynı işlecin birden fazla kullanıldığı örneklere de bu kural uygulanmaktadır (örn. A/B/C).

İşleç önceliği tanımlanmış bir gramerde işleç birleşikliği de doğru bir şekilde uygulanır.

Örn. 4'teki gramerle türetilmiş A=B+C+A cümlesini alalım. Ayrıştırma ağacı **Şekil 4**'teki gibi olur. Bu ağaca göre soldaki + işlemi, daha aşağı düzeyde kaldığı için öncelik ondadır. Önceliği aynı olan işlemler soldan sağa doğru yapılır. Bu durumda bu gramerin tanımladığı işleç birleşikliği doğru sıradadır. Matematikte toplama işleminin birleşme özelliği vardır. (A+B)+C=A+(B+C)
Ancak bilgisayarda float sayıların toplanması birleşme özelliğine sahip değildir. Çok büyük sayılara çok küçük sayılar eklerken sorun olmasa da küçük sayıları biraraya toplayıp daha sonra büyük sayıya eklediğimizde üzerinde değişiklik yapılacak basamak sayısı değiştiğinden bilgisayar için farklı anlam ifade eder. Çıkarma ve bölme işlemleri için ne matematikte ne de bilgisayarda zaten birleşme özelliği yoktur. Bunları içeren ifadelerde doğru birleşimin önemi büyüktür.



Şekil 3. Örn. 4 teki gramerle elde edilen A= B+C+A cümlesi için ayrıştırma ağacı

Bir kuralda okun sol tarafındaki bir non-terminal, okun sağındaki ifadelerin herhangi birinin başında da bulunuyorsa kural sol özyinelemeli olarak adlandırılır. Sol özyineleme soldan birleşimi belirtir. Örnekte hem toplama hem de çarpma işlemi soldan birleşimlidir (sol taraf önce hesaplanır).
Üs alma işlemi sağdan birleşimli bir işlemdir. Sağdan birleşimi gerçekleyebilmek için sağ özyineleme kullanılabilir. Sol taraftaki bir non-terminal sağ taraftaki tanımlamalardan herhangi birinin sağında yer alıyorsa o gramer kuralı sağ özyinelemelidir. Aşağıdaki örnekte **<factor>** non-terminali, okun sağındaki <exp> ** **<factor>** ifadesinin sağ tarafında bulunmaktadır.

Örn.:

<factor> → <exp> ** **<factor>** | <exp>
<exp> → (<expr>) | id

Üs alma işlemi bu tip kurallarla sağdan birleşmeli olarak tanımlanabilir (** üs alma işleminin opertörüdür).