3. Üçüncü Bölüm

3.1. Sözdizim (syntax) ve Anlambilim (Semantics)

Her programlama dilindeki geçerli programları belirleyen bir dizi kural vardır. Bu kurallar sözdizimi (syntax) ve anlambilim (semantics) olarak ikiye ayrılır.

Her deyimin sonunda noktalı virgül bulunması sözdizimi kurallarına örnek oluştururken, bir değişkenin kullanılmadan önce tanımlanması bir anlam kuralı örneğidir.

Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile **meta language (üst dil)** adı verilir.

Bu derste programlama dillerinin sözdizimini anlatmak için BNF (Backus-Naur Form) adlı üst dil kullanılacaktır. Öte yandan, anlam tanımlama için böyle bir dil bulunmamaktadır.

3.2. Sözdizim ve Anlambilim arasındaki fark

Sözdizim ve anlam arasındaki farkı, programlama dillerinden bağımsız olarak bir örnekle incelersek:

Bir tarih örneği üzerinden sözdizim ve anlam arasındaki ilişkiye bakalım

Sözdizim ve Anlam

10.06.2007 bir tarih ifadesidir fakat bu tarih ifadesi Türkiye için farklı ABD için farklı bir anlama sahiptir. 10.06.2007 Türkiye'de 10 Haziran 2007 olarak, ABD'de ise 6 Ekim 2007 olarak anlaşılır. Dolayısıyla aynı sözdizime sahip olan bir bilgi farklı anlamlara gelebilir. Bir ifade sözdizim açısından doğru olsa bile anlam açısından doğru olmayabilir.

Sözdizimindeki küçük farklar anlamda büyük farklılıklara neden olabilir. Bunlara dikkat etmek gerekir:

```
while (i<10)  while (i<10)
{ a[i]= ++i;}{ a[i]= i++;}
```

3.3. Sözdizimsel analiz

3.3.1. Soyut Sözdizim

Bir dilin soyut sözdizimi, o dilde bulunan her yapıdaki anlamlı bileşenleri tanımlar. Örneğin; +ab prefix ifadesi, a+b infix ifadesi ve ab+ postfix ifadesinde + işlemcisi ve a ve b alt-ifadelerinden oluşan aynı anlamlı bileşenleri içermektedir. Bu nedenle ağaç olarak üçünün de gösterimi yandaki şekildeki gibidir (bknz. Bölüm 2 ders slaytları)

Soyut Sözdizim Ağaçları
Bir ifadedeki işlemci/işlenen yapısını gösteren ağaçlara soyut sözdizim ağaçları adı verilir. Soyut sözdizim ağaçları, bir (a b) ifadenin yazıldığı gösterimden bağımsız olarak sözdizimsel yapısını gösterebilmeleri nedeniyle bu şekilde isimlendirilirler.

Soyut sözdizim ağaçları, uygun işlemcilerin geliştirilmesiyle diğer yapılar için de genişletilebilir.

Örneğin
if a > b then a else b

sonuc = 2* say + 18;

3.3.2. Metinsel Sözdizim

Hem doğal diller (Türkçe) hem de programlama dilleri (Java), bir alfabedeki karakter dizilerinden oluşurlar. *Bir dilin karakter dizilerine cümle veya deyim adı verilir*. Bir dilin sözdizim kuralları, o dilin alfabesinden hangi karakter dizilerinin o dilde bulunduklarını belirlerler. En büyük ve en karmaşık programlama dili bile sözdizimsel olarak çok basittir.

Bir programlama dilindeki en düşük düzeyli sözdizimsel birimlere sözlükbirim (lexeme) adı verilir. Programlar, karakterler yerine sözlükbirimler dizisi olarak düşünülebilir. *Bir dildeki sözlükbirimlerin gruplanması ile dile ilişkin andaçlar (token) oluşturulur.*

Bir programlama dilinin metinsel sözdizimi, andaçlar (token) ile tanımlanır.

Örneğin bir tanımlayıcı; toplam veya sonuc gibi sözlükbirimleri olabilen bir andaçdır. Bazı durumlarda, bir andaçın sadece tek bir olası sözlükbirimi vardır. Örneğin, toplama_işlemcisi denilen aritmetik işlemci "+" sembolü için, tek bir olası sözlükbirim vardır.

Boşluk (space), ara (tab) veya yeni satır karakterleri, andaçlar arasına yerleştirildiğinde bir programın anlamı değişmez.

Yandaki örnekte, verilen C deyimi için sözlükbirim ve andaçlar listelenmiştir. (bknz. Ders slaytları)

3.3.3. Programlama Dillerinde Gramer

Gramer, bir programlama dilinin metinsel sözdizimini açıklamak için kullanılan bir gösterimdir. Gramerler, anahtar kelimelerin ve noktalama işaretlerinin yerleri gibi metinsel ayrıntılar da dâhil olmak üzere, bir dizi kuraldan oluşur.

BNF: Backus-Naur Form

BNF (Backus-Naur Form), 1950'li yıllarda çeşitli gruplar tarafından yapılan çalışmaların sonucu olarak geliştirilen ve 1960 yılından beri programlama dilleri için standart olarak kullanılan metadildir. BNF kullanılarak sözdizimi tanımlanan ilk

programlama dili ALGOL60'tır. Daha sonraları BNF'e yapılan eklemelerle oluşan dil ise genişletilmiş BNF (extended BNF) olarak adlandırılmıştır.

BNF ile açıklanan bir gramer dört bölümden oluşur. Başlangıç (amaç) sembolü, terminal semboller, terminal_olmayan semboller ve kurallar'dan oluşan BNF gramer yapısı aşağıdaki şekildeki gibi gösterilebilir:

Gramer

BNF kullanılarak, bir dilde yer alan cümleler oluşturulabilir. Bu amaçla, başlangıç sembolünden başlayarak, dilin kurallarının sıra ile uygulanması gereklidir. Bu şekilde cümle oluşturulmasına türetme (derivation) denir ve BNF türetmeli bir yöntem olarak nitelendirilir.

Örnek gramer animasyonunda görülen dilin, atama görevini gören tek bir deyimi vardır. Bir program, begin ile başlar, end ile biter.

Ayrıştırma Ağaçları

Gramerler, tanımladıkları dilin cümlelerinin hiyerarşik sözdizimsel yapısını tarifleyebilirler. Bu hiyerarşik yapılara ayrıştırma (parse) ağaçları denir. Bir ayrıştırma ağacının en aşağıdaki düğümlerinde terminal semboller veya andaçlar yer alır. Ayrıştırma ağacının diğer düğümleri, dil yapılarını gösteren terminal olmayanları içerir.

Ayrıştırma ağaçları ve türetmeler birbirleriyle ilişkili olup, birbirlerinden türetilebilirler. Aşağıdaki şekilde yer alan ayrıştırma ağacı, "Gramerler ve Türetmeler" bölümünde tanımlanan gramere göre, bir önceki sayfada türetilmiş olan "örnek türetme" deyiminin, "x := y + z", yapısını göstermektedir.

Genişletilmiş BNF

BNF'nin okunabilirliğini ve yazılabilirliğini artırmak amacıyla, BNF'e bazı eklemeler yapılmış ve yenilenmiş BNF sürümlerine genişletilmiş BNF veya kısaca EBNF adı verilmiştir. EBNF'te Seçimlik (optionality), Yineleme (repetition) ve Değiştirme (alternation) olmak üzere üç özellik yer almaktadır:

BNF devam
Bir kuralın sağ tarafında, istenilen sayıda yinelenebilecek veya hiç yer almayabilecek bir bölümü göstermek için { } kullanımı eklenmiştir.

Bir grup içinden tek bir eleman seçilmesi gerektiği zaman seçenekler, parantezler içinde birbirlerinden "veya" işlemcisi "|" ile ayrılarak yazılabilir. Aşağıda pascal örneği verilmiştir.

<for_deyimi> -> for <değişken> := <ifade> (to | down to) <ifade> do <deyim> ;

BNF uygulaması
Temel Programlama Elemanları

Geleneksel bilgisayar mimarisi von Neumann mimarisi olarak adlandırılır. Bu mimari, her bellek hücresinin özgün bir adres ile tanımlandığı ana bellek kavramına dayanmaktadır. Bir bellek hücresinin içeriği, bir değerin belirli bir yönteme göre kodlanmış gösterimidir. Bu içerik, programların çalışması sırasında okunabilir ve değiştirilebilir.

Imperative programlama, von Neumann mimarisindeki bilgisayarlara uygun olarak programların işlem deyimleri ile bellekteki değerleri değiştirmesine dayanır.

Atama işlemi

Buyurgan (Imperative) programlamada en temel işlem atama işlemidir. Atama sembolü programlama dillerinde farklı şekilde gösterilebilir, ancak tüm programlama dillerinde atama sembolünün anlamı, sağ taraftaki değerin sol taraftaki değişkene aktarılmasıdır.

sum = 0; java, C, C++ vs.
sum := 0; pascal, algol, vs.

Değişkenler (identifiers, names)

l-value: Değişkenin adresidir

r-value: Değişkenin değeridir.

İsimler

İsimler, programlama dillerinde, değişkenlerin yanı sıra, etiketler, altprogramlar, parametreler gibi program elemanlarını tanımlamak için kullanılırlar.

İsimleri tasarlamak için programlama dillerinde farklı yaklaşımlar uygulanmaktadır.

en fazla uzunluk

büyük küçük harf duyarlılığı

özel kelimeler

İsimler – en fazla uzunluk

Programlama dillerinde bir ismin en fazla kaç karakter uzunluğunda olabileceği konusunda farklı yaklaşımlar uygulanmıştır. Önceleri programlama dillerinde bir isim için izin verilen karakter sayısı daha sınırlı iken, günümüzdeki yaklaşım, en fazla uzunluğu kullanışlı bir sayıyla sınırlamak ve çoklu isimler oluşturmak için altçizgi "_" karakterini kullanmaktır.

örnek: cok_uzun_bir_degisken_olabilir_ama_yine_de_kisa_mi

70 ve 80 li yıllarda moda olan altçizgi karakteri günümüzde yerini "camel" notasyonu tabir edilen bir yönteme bırakmıştır.

örnek: cokUzunBirDegiskenOlabilirAmaYineDeKisaMi

Dil örnekleri

FORTRAN I: maksimum 6

COBOL: maksimum 30

FORTRAN 90 ve ANSI C: maksimum 31

Ada ve Java: limit yok, bütün karakterler kullanılıyor

C++: limit yok ama derleyici hazırlayanların koydukları limitler var

LISP: limit yok

Küçük-Büyük Harf Duyarlılığı (Case Sensitivity)

Birçok programlama dilinde, isimler için kullanılan küçük ve büyük harfler arasında ayrım yapılmazken, bazı programlama dilleri (Örneğin; C, Java) isimlerde küçük-büyük harf duyarlılığını uygulamaktadır. Bu durumda, aynı harflerden oluşmuş isimler derleyici tarafından farklı olarak algılanmaktadır.

Dezavantaj: okunabilirlik (birbirine benzeyen isimler fakat farklılar)

C++ ve Java da daha kötü çünkü önceden tanımlanmış böyle isimler var (örneğin IndexOutOfBoundsException)

C, C++, ve Java isimleri büyük-küçük harf duyarlı

Fortran I-77: Harfler sadece büyük. 77'de okunurken küçük varsa büyütülüyor.

Diğer dillerde büyük küçük farkı yok.

Özel Kelimeler

Özel kelimeler, bir programlama dilindeki temel yapılar tarafından kullanılan kelimeleri göstermektedir.

Anahtar Kelime: Bir anahtar kelime (keyword), bir programlama dilinin sadece belirli içeriklerde özel anlam taşıyan kelimelerini göstermektedir. Örneğin FORTRAN'da REAL kelimesi, bir deyimin başında yer alıp, bir isim tarafından izlenirse, o deyimin tanımlama deyimi olduğunu gösterir. REAL ELMA gibi. Eğer REAL kelimesi, atama işlemcisi "=" tarafından izlenirse, bir değişken ismi olarak görülür. REAL = 87.6 gibi. Bu durum dilin okunabilirliğini azaltır.

Ayrılmış Kelime *(reserved word), bir programlama dilinde bir isim olarak kullanılamayacak özel kelimeleri göstermektedir*. Örneğin Pascal'da, for, begin, end

gibi kelimeler, C'de int, float, double gibi kelimeler isim olarak kullanılamaz ve ayrılmış kelime olarak nitelendirilir.

Özel Kelimeler

Bazı dillerde de önceden tanımlanmış kelimelerin değiştirilmesine izin verilebilir. Örneğin Ada dilinde Integer ve Float önceden tanımlanmıştır. Ancak program içinde yeniden tanımlanabilirler.

Veri Tipi

Bir veri tipi, aynı işlemlerin tanımlı olduğu değerler kümesini göstermektedir.

Bir değişkenin tipi, değişkenin tutabileceği değerleri ve o değerlere uygulanabilecek işlemleri gösterir. Örneğin; tamsayı (integer) tipi, dile bağımlı olarak belirlenen en küçük ve en büyük değerler arasında tamsayılar içerebilir ve sayısal işlemlerde yer alabilir.

Veri tipleri, programlama dillerinde önemli gelişmelerin gerçekleştiği bir alan olmuş ve bunun sonucu olarak, programlama dillerinde çeşitli veri tipleri tanıtılmıştır. Tipler temel (primitive) ve yapısal (composite) olarak gruplandırılabilir.

Temel tipler, çoğu programlama dilinde yer alan ve diğer tiplerden oluşmamış veri tiplerini göstermektedir.

C Ada

boolean Boolean = {false, true}

char Character = {. . . , 'a', . . . , 'z', . . . , '0', . . . , '9', . . . , '?', . . .}

int Integer = {. . . ,−2,−1, 0,+1,+2, . . .} -Æ {−2 147 483 648, . . . ,+2 147 483 647}

float Float = {. . . ,−1.0, . . . , 0.0, . . . ,+1.0, . . .}

Yapısal tipler ise çeşitli veri tiplerinde olabilen bileşenlerden oluşmuştur. Bir yapısal tipin elemanları, tipin bileşenlerini oluşturmaktadır. Bir yapısal tipteki her bileşenin, tip ve değer özellikleri bulunmaktadır.

Temel tipler

Bütün dilerde Boolean farklı bir veri tipi değildir. Örneğin, C++ 'da bool vardır fakat bunlar "small integer"dır. 0 false, diğer sayılar true anlamına gelir.

Bütün dilerde ayrı bir Character tipi yoktur. Örneğin C, C++ ve JAVA'da char, tipi vardır ama bunlar aslında "small integer" olurlar; içerde aralarında fark yoktur.

Bazı dillerde birden çok "integer" temel tipi vardır. Örneğin Java'da

byte {−128, . . . ,+127},
short {−32 768, . . . ,+32 767},
int {−2 147 483 648, . . . ,+2 147 483 647},

long {−9 223 372 036 854 775 808, . . . ,+9 223 372 036 854 775 807}.

Bazı dillerde birden çok gerçel sayı tipi vardır. Örneğin C, C++ ve JAVA'da float ve double..

Yapısal tiplere örnekler

Ada örneği

```
type Month is (jan, feb, mar, apr, may, jun,jul, aug, sep, oct, nov, dec);

type Day_Number is range 1 .. 31;

type Date is
        record
                m: Month;
                d: Day_Number;
        end record;
```

Java Örneği

```
enum Month {jan, feb, mar, apr, may, jun, jul, aug, sep, oct, nov, dec};
struct Date { Month m; byte d;};
```

Sabitler

Bir sabit, belirli bir tipteki bir değerin kodlanmış gösterimini içeren ancak programın çalıştırılması sırasında değiştirilemeyen değerlere verilen isimdir. Bir sabit genellikle ilkel tipte bir değerdir. Örneğin 568, bir tamsayı sabittir.

Bir değişken, bir bellek yerine bağlandığında bir değere de bağlanıyorsa ve daha sonra bu değer değiştirilemiyorsa o değişkene isimlendirilmiş sabit denir. Sabit bir değerin programda birçok kez yinelenmesi durumunda, isimlendirilmiş sabitlerin kullanılması yararlıdır. Örneğin 3.14159 değeri yerine pi isminin kullanılması, programın okunabilirliğini artırır.

Bir başka örnek olarak, 50 elemanlı bir diziyi işleyen bir programı düşünelim. Bu programda birçok kez (örneğin, dizi tanımlamada, döngülerde vb.) dizi sınırına başvuru yer alır. Bu değerin programın başında isimlendirilmiş sabit olarak tanımlanması, programın okunabilirliğini ve güvenilirliğini artırır.

İsimlendirilmiş sabitlerin tanımlanması için, Pascal'da const tanımlayıcısı kullanılır. C'de ise isimlendirilmiş sabit tanımlamak için #define kullanılır.

İşlemcilerin genel özellikleri

İşlemcilerin genel özellikleri işlenen sayısı, işlemcinin yeri, öncelik ve birleşmelilik (associativity) olmak üzere dört tanedir.

İşlenen Sayısı (arity) : Bir işlemci, alabileceği işlenen sayısına göre tekli (unary), ikili(binary) veya çoklu (nary) olabilir.

```
p = &a;        (C – unary, prefix)

i = -5;(C – unary, prefix)

i  = j – 5;     (C – binary, infix)

i = i + j;      (C – binary, infix)

i = i + j + 5;(C – binary, infix)

(plus i j 5)    (LISP – nary, prefix)

i++;   (C – unary, postfix)
```

İşlemcinin Yeri: Çoğu işlemci işlenenleri arasına yazılmakla birlikte, bazı işlemciler, işlenenlerinden önce veya sonra da yazılabilirler.

İşlemciler bir ifadede, işlenenlerden önce(prefix), işlenenler arasında (infix) ve işlenenlerden sonra (postfix) olmak üzere üç şekilde yer alabilirler.

İşlemcilerin öncelikleri

İşlemcilerin öncelikleri, birden çok işlemcinin yer aldığı bir ifadede, parantez kullanılmadığında, bir ifadenin bileşenlerinin değerlendirilme sırasını belirler.

İkili işlemciler, infix gösterimde, "a+b" de olduğu gibi işlenenleri arasına yazılır. Ancak infix gösterimdeki sorun, birden çok işlemcinin birlikte yer aldığı bir ifadede görülür. Örneğin; "a+b*c" gibi bir ifadenin değerlendirilmesi nasıl olacaktır? Sonuç, "a" ve "b*c" nin toplamı mı yoksa "a+b" ve "c" nin çarpımı mıdır? Bu soruların yanıtları işlemcilerin öncelik ve birleşmelilik kavramları ile açıklanabilir.

Her işlemcinin, programlama dili tasarlanırken önceden belirlenmiş bir önceliği vardır. Daha yüksek bir öncelik düzeyinde yer alan bir işlemci, işlenenlerini daha düşük bir düzeydeki bir işlemciden önce alır. Geleneksel bir kural olarak, sayısal işlemcilerden "*" ve "/", toplama "+" ve çıkarmadan "-" daha yüksek önceliğe sahiptir. Bu nedenle yukarıdaki ifadede, "*" işlemcisi, işlenenlerini "+" dan önce alır ve "a+b*c", "a+(b*c)" ye eşittir.

Birleşmelilik : *Bir ifadede aynı öncelik düzeyinde iki işlemci bulunuyorsa, hangi işlemcinin önce değerlendirileceği dilin birleşmelilik kuralları ile belirlenir*. Bir işlemci, sağ veya sol birleşmeli olabilir.

Sol birleşmeli: Bir işlemcinin birden çok kez yer aldığı bir ifadedeki alt ifadeler, soldan sağa olarak gruplanırsa, işlemci sol birleşmeli olarak adlandırılır. Aritmetik işlemcilerden "+","-","*" ve "/" sol birleşmelidir.

Sağ birleşmeli: Öte yandan, eğer bir işlemcinin birden çok kez yer aldığı bir ifadedeki alt ifadeler, sağdan sola gruplanırsa işlemci, sağ birleşmeli olarak adlandırılır. Üs alma işlemcisi sağ birleşmelidir.

Niteliğine Göre İşlemciler

İşlemciler, işlenenlerin niteliğine göre sayısal, ilişkisel veya mantıksal işlemciler olabilirler.

Sayısal işlemciler, sayısal işlenenlere uygulanan işlemcilerdir. Kullanılan semboller programlama dillerinde farklılık gösterebilmekle birlikte, genel olarak üs alma, toplama, çıkarma, mod, çarpma, bölme gibi işlemcilerdir.

Bir ilişkisel işlemci, iki işleneninin değerlerini karşılaştırır ve eğer mantıksal (Boolean) tipi dilde tanımlı bir veri tipi ise mantıksal tipte bir sonuç oluşturur.

Genellikle, ilişkisel işlemcilerle kullanılabilen işlenenler, sayısal, karakter veya sıralı (ordinal) tiplerde olurlar.

Mantıksal işlemciler, sadece mantıksal (Boolean) işlenenleri alırlar ve mantıksal değerler, DOĞRU ve YANLIŞ üretirler.

Mantıksal işlemciler, AND (ve), OR(veya), NOT (değil) XOR(dışlayan veya) gibi işlemleri de içerebilirler. Mantıksal işlemcilerde öncelik sıralaması genellikle NOT, AND ve OR şeklindedir.

İşlemci yükleme

İşlemcilerin anlamlarının, işlenenlerin sayısına ve tipine bağlı olarak belirlenmesine işlemci yüklemesi (operator overloading) denir.

Sayısal işlemciler, programlama dillerinde sıklıkla birden çok anlamda kullanılırlar. Örneğin "+", hem tamsayı hem de kayan-noktalı toplama için kullanılır ve bazı dillerde, sayısal işlemlere ek olarak karakter dizgilerin birleştirilmesi için de kullanılır. İşlemci yüklemelerinde, "+" da olduğu gibi, benzer anlamlarda olmayabilir.

Bir diğer örnek olan '-' işlemcisi, hem bir sayısal değerin negatif olduğunu belirtmek için tekli işlemci olarak, hem de ikili bir işlemci olan sayısal çıkarma işlemini göstermek için kullanılır. Her ne kadar işlemcinin iki kullanımında anlamsal yakınlık varsa da, bir ifadede yanlışlıkla birinci işlenenin unutulması, derleyici tarafından hata olarak algılanmayacak ve ikinci işlenen negatif değer olarak kabul edilecektir. Bu ve benzeri işlemci yüklemeleri, fark edilmesi güç hatalara neden olabilmektedir.

Deyimler

Deyimler, bir programdaki işlemleri göstermek ve akışı yönlendirmek için kullanılan yapılardır. Deyimler basit veya birleşik olabilirler. Basit deyimlere örnek olarak atama deyimi verilebilir. Birleşik deyimler ise bir dizi deyimin tek bir deyime soyutlanmasını sağlarlar. Birleşik bir deyimde yer alan deyimleri belirlemek için basit deyimlerden ayrı

bir sözdizime gereksinim vardır. Örneğin Pascal'da, birleşik deyimler begin ve end anahtar kelimeleri arasında, C de "{}" parantezleri arasında gruplanır.

Programlarda akışı yönlendirmek için seçimli deyimler (if-then- else, case deyimleri gibi) ve yinelemeli deyimler (while, for deyimleri gibi) kullanılabilir.

Altprogramlar, bir dizi deyimin gruplanmasını ve bir isim ile gösterilmesini sağlarlar. Altprogramlar, bir başlık, yerel tanımlamalar bölümü ve işlemlerin yer aldığı bir gövde ile tanımlanır. Altprogram başlığı, altprogram ismini ve varsa tipleriyle birlikte altprogramın parametrelerini belirtir. Altprogram gövdesi, altprogram etkin olduğunda çalıştırılacak deyimlerden oluşur.

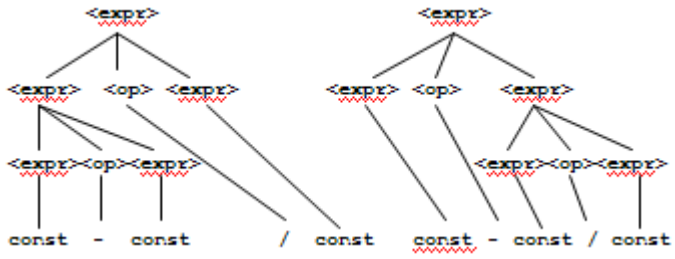
Altprogramlar ve parametre aktarımları daha sonra incelenecektir.

Muğlâklık ve muğlâklığın ortadan kaldırılması

Bir muğlâk ifade grameri:

<expr> -> <expr> <op> <expr> | const
<op> -> / | -

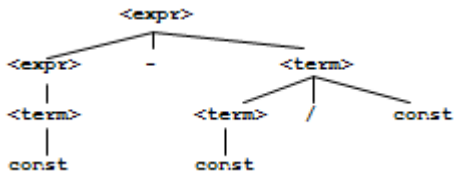
Şeklinde verilebilecek örnek gramer için operatör seviyeleri sebebiyle bir muğlâklık (ambigous) söz konusudur.



Bu tip muğlâk durumları ortadan kaldırmak için ayrıştırma ağacı kullanılır. **Eğer biz işlemlerin emsal seviyelerini azaltmak için ayrıştırma ağacı kullanırsak muğlâklık ortadan kalkar.**

Bir muğlâk olmayan ifade grameri:

<expr> -> <expr> - <term> | <term>
<term> -> <term> / const | const



Extended BNF (EBNF)

BNF gramer yapısının genişletilmesi sonucu EBNF elde edilmiştir. EBNF ile sağlanan kısaltmalar aşağıda verilmiştir.

1. Seçimlik parçalar köşeli parantezler içinde yer alır ([])
   <proc_call> -> ident [(<expr_list>)]
2. (Kuralın sağ tarafı) RHS'nin alternatif kısımları parantezler içine konur ve onlar dikey çubuklarla ayrılır.

<term> -> <term> (+ | -) const

3. Tekrarlar (0 veya daha fazla) küme parantezleri ({}) ile verilir
<ident> -> letter {letter | digit}

BNF:

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

EBNF:

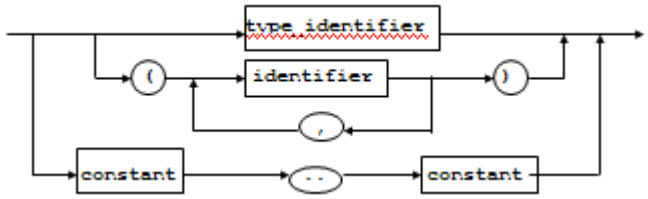
<expr> -> <term> {(+ | -) <term>}
<term> -> <factor> {(* | /) <factor>}

Sözdizimi Grafikleri (Syntax Graphs)

Bir grafik; düğümlerin ve kenarlar olarak isimlendirilen bazı çizgilerin bir toplamıdır. Yönlü bir grafik (directed graph) kenarları yönlendirilmiş bir grafik türüdür ve okların yönünün bir anlamı vardır.

BNF ve EBNF kuralları bir yönlü grafik ile sunulabilir, böylesi grafikler sözdizim grafikleri olarak adlandırılır. Bunlara aynı zamanda sözdizim diyagramları veya sözdizim çizimleri de denir.

Sözdizim grafikleri bir gramerin kurallarının sağ tarafındaki terminal ve terminal olmayan sembollerin sunumu için farklı düğümler kullanırlar. Dikdörtgen düğümler sözcüksel birimlerin (non terminals) isimlerini içerir. Daire ve elipsler terminal sembolleri içerir.



Pascal tipi bildirimler için örnek sözdizimi grafiği

Özyinelemeli Ayrıştırma (Recursive Descent Parsing)

Ayrıştırma, verilen bir giriş katarından bir ağacın izinin sürülmesi veya bir ayrıştırma ağacının inşa edilmesidir. Ayrıştırıcılar genellikle bir (sözcüksel) lexical analizci tarafından üretilen sözlük birimleri analiz etmez

Bir *recursive descent parser* bir ayrıştırma ağacını yukarıdan aşağıya sırasında izler; o bir yukarıdan aşağıya ayrıştırıcıdır. Gramerde her bir terminal olmayan bileşenin kendi ile uyumlu bir altprogramı vardır; altprogram terminal olmayanı üretebilen bütün sentential formları ayrıştırır.

Recursive descent parsing altprogramları gramer kurallarından doğrudan inşa edilir.

<term> -> <factor> {(* | /) <factor>}

Örnekte verilen gramer için takip eden recursive descent parsing alt programını kullanabiliriz. (Parser C'de yazılmıştır)

```
void term() {
  factor();  /* ilk faktörü ayrıştır */
  while (next_token == ast_code ||
         next_token == slash_code) {
    lexical();  /* sıradaki tokenı al */
    factor();  /* sıradaki faktörü ayrıştır */
  }
}
```

3.4. Semantik Analiz

3.4.1. Statik Semantikler

Tip uyumluluğu kuralları gibi kuralların BNF ile açıklanabilmesi zordur. Bu ve buna benzer durumlar statik semantik kuralları ile açıklanırlar.

Öznitelik Gramerleri (Attribute Grammers)

Bir öznitelik grameri bir dilin yapısını Context Free Grammer (CFG) den daha iyi açıklayan bir araçtır. Öznitelik grameri CFG için iyi bir ektir. Tip uyumluluğu gibi konuları çözmede kullanılır.

Dinamik Semantikler

Dinamik semantikler konusuna; deyimler, ifadeler ve program parçalarının anlamı konuları girmektedir. Dinamik semantikler için uluslar arası kabul görmüş bir notasyon bulunmamaktadır.

Operasyonel Semantikler

Konuyu anlamak için bir komutun çalıştırılmasını düşünelim. Komut çalışmadan önce makinenin registerleri ve bellek lokasyonları farklı değerler tutarken komut sonrasında register ve bellek lokasyonları farklı değerler tutacaktır. Operasyonel semantiklerin konumu işte bu değişimin anlaşılmasıdır.

Temel Proses

Operasyonel semantikleri anlamak için yüksek seviyeli bir dilde yazılmış ifadeler daha düşük seviyeli bir dile çevrilir. Örneğin bir programın C kodlaması ve düşük dildeki karşılığı aşağıdaki şekildedir.

C ifadesi	Operasyonel semantikler

For(expr1;expr2;expr3)
{
…
}
	Expr1;
	Loop: if expr2=0 goto out
	…
	Goto loop
	Out: …

Değerlendirme
Formal operasyonel semantikler için kullanılan ilk ve en önemli örnek PL/I için kullanılan VDL isimli tanımlama dilidir.

Aksiyomatik Semantikler

Aksiyomatik semantikler programların doğruluğunu ispat için bir yöntemin geliştirilmesi ile ilgilidir.

Assertions

Aksiyomatik semantikler matematiksel mantık tabanlıdır. *Mantıksal deyimler assertions olarak adlandırılır*. Assertions precondition veya postcondition olarak adlandırılır.

Sum=2*x+1 {sum>1}
Bu bir ifade ve postcondition'dur. {x>10} ise bu ifade için precondition olabilir.

Weeakest precondition en azından uyumlu postcondition'un doğruluğunu garanti eden preconditiondur. Örneğin yukarıdaki ifade ve postcondition için {x>10}, {x>50}, {x>1000} hepside geçerli precondition'dur. Bu preconditionlar için weakest precondition {x>0} dır.

Denotational Semantikler

Özyinelemeli fonksiyon teorisi tabanlıdır. Soyut semantik açıklama metodunun en sık kullanıldığı yöntemdir.
Bir dil için denotational spec inşa etme prosesi

1. Her bir dil varlığı için matematiksel bir nesne tanımlar
2. Dilin varlıklarını matematiksel nesnelerin örneklerine uygun olarak eşleştiren bir fonksiyon tanımlayın.

Dilin anlamı sadece program değişkenlerinin değerleri ile tanımlanır.
Operasyonel ve denotational semantikler arasındaki farklar: operasyonel semantiklerde kodlanmış algoritmalarla durum değişimleri tanımlanır. Denotational semantiklerde ise onlar matematiksel fonksiyonlarla tanımlanır.

Bir programın durumu onun bütün geçerli değişkenlerinin değerleridir.
S={<i1,v1>, <i2,v2>,…, <in,vn>}

VARMAP öyle bir fonksiyon olsun ki ona bir değişken adı ve bir durum verildiği zaman geriye değişkenin geçerli değeri dönsün.
VARMAP(ij, s)= vj

Denotational semantikler

- Programların doğruluğunu ispat için kullanılabilir
- Programlar hakkında düşünmek için bir gösterim sunar
- Derleyici üretim sistemlerinde kullanılabilir.