BLG339
PROGRAMLAMA DİLLERİ KAVRAMI

Hafta 2

Yrd. Doç. Dr. Melike Şah Direkoğlu

Alındığı kaynak:
Addison-Wesley's Programming Language Concepts slaytları ve Prof. Dr. Tuğrul Yılmaz' ın ders notlarından faydalanarak hazırlanmıştır.

Konular

- Sözdizim (syntax) ve Anlambilim (Semantics) Analizi
- Sözdizim(Syntax) Tanımlamanın Genel Terminolojisi
- Anlambilim (Semantics) Tanımlamanın Genel Terminolojisi

Sözdizim (Syntax) ve Anlambilim (Semantics)

- Her programlama dilindeki geçerli programları belirleyen bir dizi kural vardır. Bu kurallar sözdizim (syntax) ve anlambilim (semantics) olarak ikiye ayrılır.
  - Her deyimin sonunda noktalı virgül bulunması sözdizim kurallarına bir örnek, bir değişkenin kullanılmadan önce tanımlanması bir anlam kuralı örneğidir.
- Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile metadil (metalanguage) adı verilir.
  - Bu derste programlama dillerinin sözdizimini anlatmak için BNF (Backus-Naur Form) adlı metadil kullanılacaktır. Öte yandan, anlam tanımlama için böyle bir dil bulunmamaktadır.

Sözdizim ve Anlambilim (devam)

- Sözdizimi (Syntax) ve anlamsallar/anlambilim (semantics) bir dilin tanımını sağlarlar.
  - Bir dil tanımının kullanıcıları
    - Diğer dil tasarımcıları; dilin nasıl çalıştığını anlamak için
    - Gerçekleştiriciler (implementors/compilers); programın nasıl çalıştığını anlamak için
    - Programcılar; nasıl programlanacağını anlamak için

Sözdizim ve Anlambilim (devam)

- Sözdizim ve anlam arasındaki farkı, programlama dillerinden bağımsız olarak bir örnekle incelersek:
- Tarih gg.aa.yyyy şeklinde gösteriliyor olsun.

Sözdizim	Anlam	
10.06.2007	10 Haziran 2007	Türkiye
	6 Ekim 2007	ABD

- Bir başka örnek;
  "while (boolean ifade) deyim"
  → semantics (anlamı): Boolean ifade doğru ise deyime geç ve boolean ifade doğru olduğu sürece dögüye devam et.
- Ayrıca sözdizimindeki küçük farklar anlamda büyük farklılıklara neden olabilir. Bunlara dikkat etmek gerekir:

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

- İyi tasarlanmış programlama dillerinde syntax (sözdizim) ve semantics (anlam) birbiriyle bağlantılıdır.

Sözdizim (Syntax)

Sözdizim(Syntax) Tanımlamanın Genel Terminilojisi

- Dil konuşma dili (Türkçe) veya yapay bir dil (Java) olsun, dil karakterler ve cümleler/deyimlerden meydana gelmektedir.
  - Karakterlerin tümü alfabeyi oluşturur.
  - Cümleler kümesi de dili oluşturur.
- Karakterin sözdizimi kuralları o alfabede kelimelerden nasıl cümleler oluştuğunu belirler.
  - Örnek: Türkçede özne başta, fiil sonda olur. Gizli özne kuralı, devrik cümle kuralı, vs. Bu kuralların hepsi Türkçe dilinin nasıl kullanılacağını belirlediği gibi, bir programlama dilinin syntax (sözdizim) kuralları da o dilin nasıl kullanılacağını belirler.
- Bir lexeme dilin en düşük seviyeli sözdizimsel birimidir
  - örn., operatörler (+, - *, /), sayısal değerler, özel kelimeler (begin, end)

Sözdizim(Syntax) Tanımlamanın Genel Terminilojisi (Devam)

- Simge(token) da Lexeme lerin bulunduğu kategoriyi temsil eder
  - Değişken (variable), metot (method) veya sınıf (class) isimlerine identifier grubu denir.
  - Her grup da bir isim veya token (simge) ile temsil edilir (örn. identifier)

token	sample lexemes
identifier	`count, i, x2, ...`
if	`if`
add_op	`+`
semi	`;`
int_lit	`0, 10, -2.4`

Lexeme ve Token Örneği

- Java deyimi:
  - toplam = 2 * sayi + 17;

Lexeme	Token (simge)
toplam	identifier
=	equal_sign
2	int_literal
*	mult_op
sayi	identifier
+	plus_op
17	int_literal
;	semicolon

Dillerin Resmi Tanımı

- Diller iki farklı method ile; recognition (tanıyıcılar) ve generation (üreticiler) ile tanımlanabilirler.
- Dil Tanıyıcıları(Recognizers)
  - Bir tanıyıcı aygıt dilden bir karakter dizisi okur ve bu karakter dizisinin bu dile ait olup olmadığına karar verir; karakter dizisisi ya kabul olunur yada reddedilir.
  - Bir dilde birçok kelime seçeneği olabileceğinden bu zaman alır. Fakat dilin tüm olası kelimelerini yaratmak için kullanılır.
  - Örnek: bir derleyicinin (compiler) söz dizim analizcisi (sadece eldeki programın sözdizimi dile uyup uymadığını test ettiğinden hızlıdır).
- Dil Üreticileri(Generators)
  - Bir dile ait cümle üreten aygıtlar olarak düşünülebilir.
  - Belirli bir cümlenin söz diziminin doğru olup olamadığı o dili üreten üretici incelenerek tespit edilebilir.
- Bu iki metot dillerin nasıl derlendiği (compile) ve çalıştığını (interpret) etkilemiştir.

Söz Diziminin Tanımında Kullanılan Resmi Metotlar

- Serbest- İçerik Gramerleri (Context-Free Grammars) aynı zamanda Backus-Naur Form olarak da bilinir.
  - Programlama dillerinin söz dizimlerini tanımlamada kullanılan en yaygın metottur.
- Genişletilmiş BNF(Extended BNF)
  - BNF nin okunabilirliğini ve yazılabilirliğini arttırır.
- Gramerler ve Dil Tanıyıcıları

BNF ve Serbest-İçerik Gramlerleri (Context- Free Grammars)

- Serbest İçerik Gramerleri (dil bilgisi)
  - Noam Chomsky (linguist) tarafından 1950 lerin ortasında doğal dillerin söz dizimlerini tanımlamak için geliştirdi
  - Serbest-İçerik dilleri adlanılan bir diller sınıfını tanımladı.
  - Serbest-İçerik dilleri (Context-free grammars) daha sonra programlama dillerini uygulandı.

Backus-Naur Form (BNF)

- BNF, 1950'li yıllarda John Backus ve Peter Naur 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.
- Metalanguage (meta dil) bir başka dili anlatmak amacıyla kullanılan dile denir. BNF bir metadildir.

Backus-Naur Form (BNF) devam

- BNF de, soyutlamalar (abstraction) kullanılarak söz dizimsel yapıların temsil edilir.
- Örnek abstraction (soyutlama): Assign (atama) kuralı

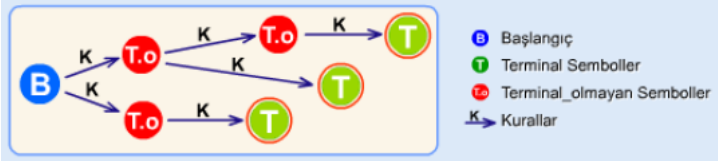
<assign> → <var> = <expression>

Left hand side (LHT) (sol taraf) Right hand side (RHT) (sağ taraf)

  - Bu bir kuraldır ve assign soyutlamasının tanımlanabilmesi için <var> ve <expression> da tanımlanması gerekir
  - Toplam = toplam1 + toplam2 buna örnektir.
  - Burada <assign> yada örnektedi toplam non-terminal symbol (son olmayan semboller) ve toplam1, toplam2 ve lexemeler (=, +) da terminal symbol (son olan sembol) olarak adlandırılır.
- BNF grameri de bir kurallar topluluğudur.

BNF Temelleri

- Son Olmayan semboller(Non-terminals): BNF soyutlamaları olarak adlandırılır (abstractions)
- Başlangıç sembolu
- Son semboller(Terminals): lexeme lar ve token lar
- Gramer: bir kurallar koleksiyonudur.
- BNF gramer yapısı aşağıdaki şekildeki gibi gösterilebilir:



BNF Kurallar

- Bir kuralda sol taraf (LHS-left hand side) ve sağ taraf(RHS-right hand side) vardır ve son olan(terminal) ve son olmayan (nonterminal) sembollerden oluşur.
- Bir gramer sonu boş olmayan kurallar kümesidir.
- Bir soyutlamanın (abstraction) (veya son olmayan sembol) birden fazla sağ tarafı olabilir.

  <stmt> → <single_stmt> | begin <stmt_list> end
- '|' işareti logical OR (veya) anlamındadır.

BNF Kurallar – Birden fazla sağ tarafın birleştirilmesi

- Java if deyimini örneği;

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

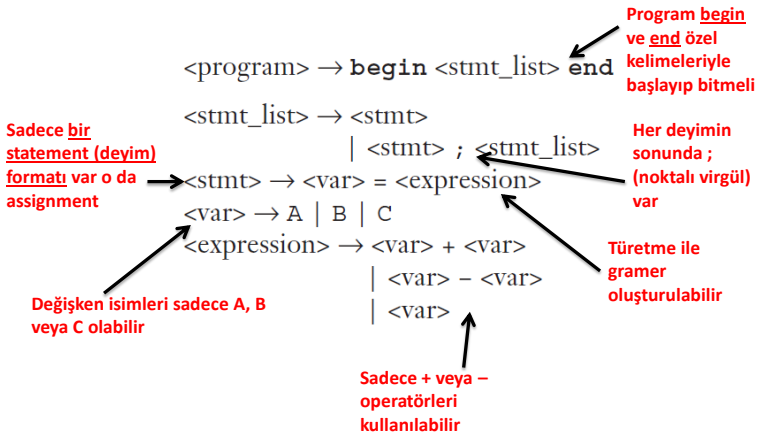
VEYA

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

BNF Kurallar – Listelerin Tanımlanması ve Gramer

- BNF'te listeleri tanımlamak için recursion (kendi kendini çağırma/yineleme) metodu kullanılır.
  <ident_list> → ident | ident, <ident_list>
- Bu türetme (derivation) kuralların tekrarlı bir şekilde uygulanmasıdır. Bir başlangıç sembolünden başlanır ve tüm elemanları son olan sembol (terminal) den oluşan bir cümle de biter.
- Türetme kullanılarak gramer oluşturulur.

Küçük bir Program Dili Gramer Örneği



Örnek Dilinden Türemiş Program

Program `begin A = B + C ; B = C end`

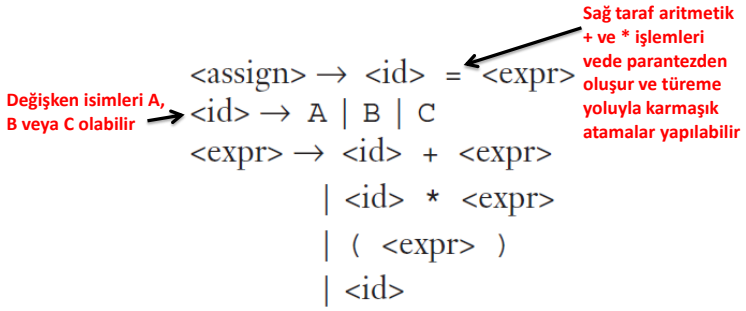
Gramer

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

Bütün türemeler program sembolu ile başlamalıdır

<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**

Basit Atama Deyimi Grameri



A = B * (A + C) deyimi bu gramere bir örnektir

Örnek Atamadan Türemiş Deyim

A = B * (A + C)

Gramer

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

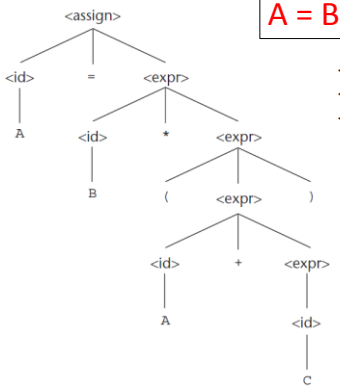
Bütün türemeler assign sembolu ile başlamalıdır

<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)

Ayrıştırma Ağacı (Parse Tree)

- Bir türetmenin hiyerarşik temsilidir.
- Bir dilin deyimlerinin (expressions) hiyerarşik olarak tanımlanması sağlar.

A = B * (A + C) deyimi



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

Gramerlerin Belirsizliği (Ambiguity)

- Bir gramerdeki bir cümlesel biçim iki veya daha fazla ayrıştırma ağacı (parse tree) oluşturuyorsa bu gramer belirsiz (ambiguous) olarak adlanır.

Bir Belirsiz Deyim(expression) Grameri

Önceki Gramer

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

Belirsiz Gramer

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

Belirsizlik, gramerin sözdizim kurallarında daha esnek olmasından kaynaklanır!

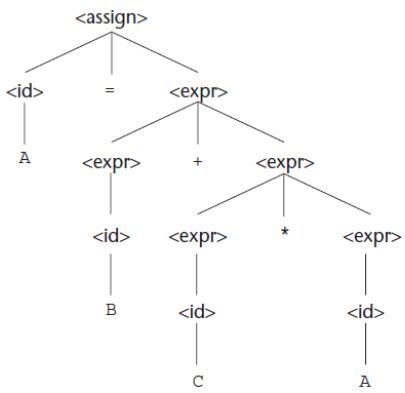
Belirsiz Deyim Ayrıştırma Ağacı (Parse Tree)

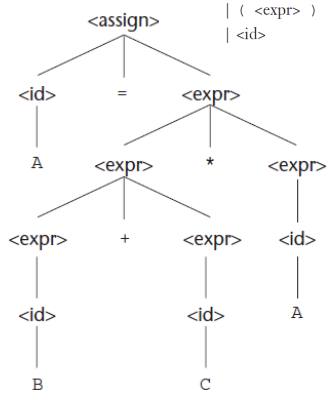
Deyim

A = B + C * A

Belirsiz Gramer

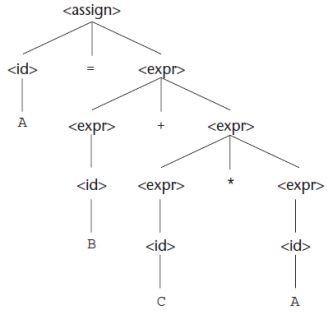
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>





Belirsiz Deyimde İşleçlerin (operator) Önceliği

- Eğer ayrıştırma ağacında işleçlerin(operators) öncelik seviyeleri gösterilirse belirsizlik olmaz.
- Bir ayrıştırma ağacında işleç (operator), ağacın (parse tree) ne kadar aşağısında yaratılıyorsa, o kadar öncelik hakkı var demektir.



Burada çarpma işlecinin öncellik hakkı var çünkü ilk bu işlem ele alınır.

Belirsiz Deyim Ayrıştırma Ağacı (Devam)

- Deyimler
  - A = A + B * C (B * C) ayrıştırma ağacında en aşağıda
  - A = A * B + C (B + C) ayrıştırma ağacında en aşağıda
    - Özetle bu gramerde operatörlerin sırası bir belirsizlik yaratıyor
- Gramerde basit deyimler kullanarak ve operatör (işleç) sırasını belirterek belirsizlik yok edilebilir.
  - Bunun için başka son olmayan semboller (non-terminals) ve kurallar gereklidir.
  - Örn. Örnekteki <expr> non-terminal (son olmayan sembol) ile birlikte çarpma (factor non-terminal)ve toplama (term non-terminal) sembolleri kullanılarak işlem sırası belirtilebilir.
  - <expr> sembolü root (ağaçta en üstte) olduğundan expr ile sadece toplama için kullanılan <term> sembolü, <expr> sembolü ile birleştirilirse, toplama her zaman ağacın üst kısmında kalmış olur. (yani çarpma önceliği eklenir)

Bir Belirsiz Olmayan (Unambigous) Deyim (Expression) Grameri

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

Toplam her zaman expr ile birlikte yapılmak zorunda olduğundan, toplam ayrıştırma ağacında üstte yer alır. Çarpma toplamaya eklendiğinden çarpma her zaman ayrıştırma ağacında en aşağıda yer alır.

Belirsiz Olmayan Gramerden Türemiş Deyim

Belirsiz Olmayan Gramer

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

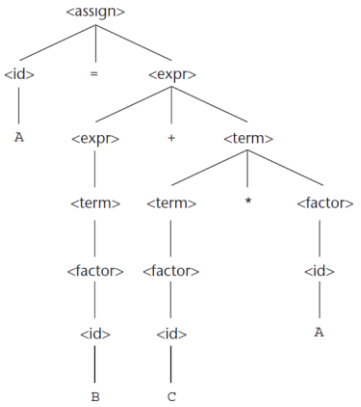
Deyim

A = B + C * A

<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

Belirsiz Olmayan Deyim Ayrıştırma Ağacı (Parse Tree)

- Ayrıştırma ağacı (parse tree) türemiş gramerden kolaylıkla çıkarılabilir.



Örnekler

- Verilen grameri kullanarak, aşağıdaki deyimler için sol taraftan türetme kullanarak, ayrıştırma ağacını (parse tree) gösteriniz.

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

a. A = A * (B + (C * A))
b. B = C * (A * C + B)
c. A = A * (B + (C))

Örnekler

- Aşağıdaki gramerin oluşturduğu dili bir cümle ile açıklayınız.

<S> → <A> <B> <C>

<A> → a <A> | a

<B> → b <B> | b

<C> → c <C> | c

Örnekler

- Verile gramer şöyledir:

<S> → <A> a <B> b

<A> → <A> b | b

<B> → a <B> | a

- Hangi cümleler bu gramerden türemiştir?
  a. baab
  b. bbbab
  c. bbaaaaa
  d. bbaab

Örnekler

- Verile gramer şöyledir:

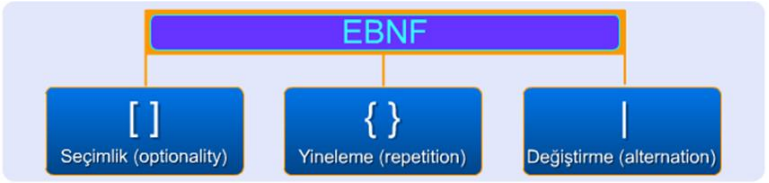
<S> → a <S> c <B> | <A> | b
<A> → c <A> | c
<B> → d | <A>

- Hangi cümleler bu gramerden türemiştir?

a. abcd
b. acccbd
c. acccbcc
d. acd
e. accc

Genişletilmiş BNF (Extended BNF)

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



EBNF – Seçim/Optionality [] ve Değiştirme/Alternation (|)

- EBNF de seçimli olan parçalar [ ] içinde tanımlanır.

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

- BNF te ise

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

- Sağ tarafın(RHS) alernatif seçenekli kısımları parantez içinde dikey çubuklar ile ayrılarak gösterilir.

<term> → <term> (* | / | %) <factor>

- BNF te ise

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

EBNF – Yineleme (Recursion)

- (0 ya da daha fazla) tekrarlamalar { } içinde yazılır; ya hiç yazılmaz (0) veya sonsuza kadar tekrarlanabilir.
  - <ident_list> → <identifier> {, <identifier>}
  - <ident> → letter {letter|digit}

Örnek BNF ve EBNF Grameri

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
```

Özellik Gramerleri (Attribute Grammars)

- Serbest içerik gramerleri programlama dillerinin tüm söz dizimini tanımlamazlar
  - Örneğin bir değişkenin tanımlanmadan önce kullanılamaması veya float bir sayının integer bir sayıya atanamaması
- Bu tür problemler genellikle static semantics (statik anlamı) olarak tanımlanır. Program anlamı ile ilgili bir konu olmasına rağmen, derleyici (compiler) tarafından, program derlenirken bu tür hatalar bulunmak zorundadır. Özetle sözdizimini (syntax) de etkiler. Fakat BNF ile bu tür anlamlar tanımlanamaz.
- Özellik gramerleri (attribute grammars) bu sorunu çözen bir serbest içerik grameridir (context-free grammar).
  - Programın hem sözdizimi hemde statik anlamını tanımlayarak, programın derlenmesini sağlarlar. İki özelliği vardır:
    - Statik anlamsal belirtimleri (static semantics specification)
    - Derleyici tasarımı (static semantics checking)

Özellik Gramerleri Tanımlama

- Her terminal (son olan sembol) ve non-terminal (son olmayan sembol) için özellikler (attributes) eklenmiştir.
- Attribute functions (veya semantic functions) olarak adlandırılan anlam fonksiyonları kurallar ile birlikte çalışır.
  - Bu şekilde dilin statik anlamsal bilgileri gramer kurallarına eklenmiş olur.
  - Bu anlam fonksiyonlarına predicate functions denir.

Özellik Gramerleri Tanımlama

- Bir özellik grameri aşağıdaki eklentilere sahip bir serbest içerik grameridir.
  - Her x gramer kuralı için bir A(x) özellik değerleri kümesi vardır.
  - Özellikler syntesized (ayrıştırma ağacına bilgi geçiren) veya inherited (ayrıştırma ağacında kalıtım yoluyla aşagıdaki bölgelere bilgi geçiren) özellikler olarak ikiye ayrılır.
  - Syntesized S(X) fonksiyonu ile
  - Inherited I(X) fonksiyonu ile gösterilir.

Özellik Gramerlerine Örnek

- Atama gramerine değişken türü anlamının eklenmesi

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

BNF Gramer

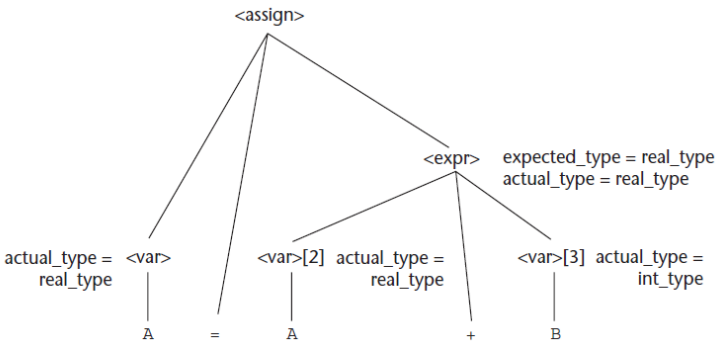
Özellik Grameri

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)

Non-terminaller isimlenmiştir

Anlam fonksiyonu (predicate function)

Özellik Gramerleri Eklenmiş Ayrıştırma Ağacı (Attributed Parse Tree)



AnlamBilim / Anlamsallar (Semantics)

Anlambilim/Anlamsallar (Semantics)

- Anlamsalları tanımlamak için geçiş çapta kabul edilen tek bir notasyon yoktur.
- İşlemsel Anlamlar(Operational Semantics)
  - Bir program deyiminin veya programın, makinede çalıştırarak anlamaya çalış; ya simüle eder ya da gerçekten çalıştırır.
    - Örnegin, basit bir print metodu için yazılmış program, makinenin durumundaki (bellek-memory, yazmaçlar-registers) değişiklik ifadenin anlamını tanımlar.

İşlemsel Anlamlar (Operational Semantics)

- Bir yüksek seviye dil için işlemsel anlamları kullanmak için bir sanal makineye ihtiyaç vardır.
- Bir dönüştürücü oluştur (kaynak kodu idealleştirilmiş bilgisayar için makine koduna dönüştür) (intermediate language)
- İdealleştirilmiş bilgisayar için bir simulatör oluştur.

```
for (expr1; expr2; expr3) {
  ...
}
```

C Kodu

```
        expr1;
loop:   if expr2 == 0 goto out
        ...
        expr3;
        goto loop
out:    ...
```

İnsanlar için anlamı

```
ident = var
ident = ident + 1
ident = ident – 1
goto label
if var relop var goto label
```

Bilgisayar simülasyonu için anlamı

Aksiyomatik Anlamlar (Axiomatic Semantics)

- Biçimsel mantık temellidir – matematiğe dayanır.
- Program değişkenleri ve program ifadeleri arasındaki anlamı anlamak icin kullanılır.
- Orijinal amaç: biçimsel program doğrulama – program biçimi doğru mu?
- Aksiyonlar veya girişim (inference) kuralları dildeki her ifade için tanımlanır (bu deyimlerden başka deyimlere dönüştürmeye izin verir)
- Deyimler iddialar(assertions) olarak adlandırılır.

Aksiyomatik Anlamlar (Axiomatic Semantics)

- Bir ifadeden önceki iddia (a precondition) değişkenler arasındaki ilişkileri ve kısıtları tanımlar
- Bir ifadeden sonraki iddia postcondition dır.
- En zayıf ön koşul (weakest precondition) en az kısıtlayıcı önkoşuldur ve bu koşul post condition ı garanti eder.
- Bir örnek
  - Mümkün önkoşul(precondition): {b > 10}
  - En zayıf önkoşul: {b > 0}