Giriş

- Her programlama dilindeki geçerli programları belirleyen bir dizi kural vardır. Bu kurallar **sentaks** (sözdizim, syntax) ve **semantik** (anlambilim, semantics) olarak ikiye ayrılır.
- Her deyimin sonunda noktalı virgül bulunması sentaks kurallarına örnek oluştururken, bir değişkenin kullanılmadan önce tanımlanması bir semantik kuralı örneğidir.

Sentaks (Sözdizimi) ve Semantik (Anlam)

- Sentaks (Syntax): İfadelerin (statements), deyimlerin (expressions), ve program birimlerinin biçimi veya yapısı
- Semantik (Semantics): Deyimlerin, ifadelerin, ve program birimlerinin anlamı
- Sentaks ve semantik bir dilin tanımını sağlar
  - Bir dil tanımının kullanıcıları
    - Diğer dil tasarımcıları
    - Uygulamacılar (Implementers)
    - Programcılar (Dilin kullanıcıları)

Sentaks (Sözdizimi) ve Semantik (Anlam)

- Sentaks (Sözdizimi) ve Semantik (Anlam) bir dilin tanımını sağlar

Sözdizim (Syntax)	Anlam (Semantics)
Bir dilin sözdizim kuralları, bir deyimdeki her kelimenin nasıl yazılabileceğini belirler.	Bir dilin anlam kuralları ise, bir program çalıştırıldığında gerçekleşecek işlemleri tanımlar.

Sentaks (Sözdizim) ve Semantik (Anlam)

- 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 6 Ekim 2007	Türkiye ABD

- Ayrıca sözdizimindeki küçük farklar anlamda büyük farklılıklara neden olabilir. Bunlara dikkat etmek gerekir:
-

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

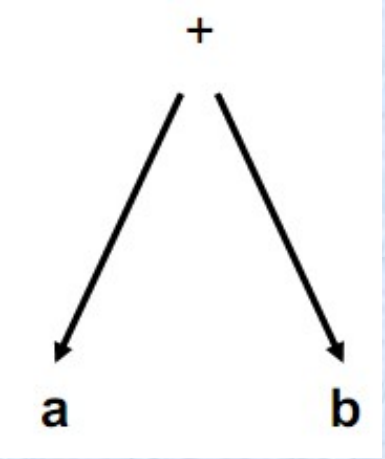
```
while (i<10)
{ a[i]= i++;}
```

Soyut Sözdizim

- Bir dilin **soyut sözdizimi**, o dilde bulunan her yapıdaki anlamlı bileşenleri tanımlar.
- Örneğin;
  - *ab prefix* ifadesi,
  - *a+b infix* ifadesi,
  - *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.

+ab prefix
a+b infix
ab+ postfix için



Metinsel Sözdizim

- Hem doğal diller hem de programlama dilleri, 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 belirler. En büyük ve en karmaşık programlama dili bile sözdizimsel olarak çok basittir.
- Bir programlama dilindeki en düşük düzeyli sözdizimsel birimlere **lexeme** adı verilir. Programlar, karakterler yerine *lexeme*'ler dizisi olarak düşünülebilir.
- Bir dildeki *lexeme*'lerin gruplanması ile dile ilişkin **token**'lar oluşturulur.

Metinsel Sözdizim

puan = 4 * dogru + 10;

Lexeme	Token
puan	Tanımlayıcı
doğru	
4	tamsayı_sabit
10	
=	eşit_işareti
*	çarpım_işlemcisi
+	toplama_işlemcisi
;	noktalı virgül

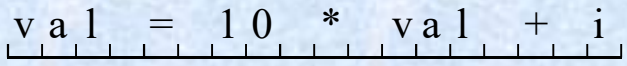
- Bir programlama dilinin metinsel sözdizimi, *token*'lar ile tanımlanır. Örneğin bir tanımlayıcı; *toplam* veya *sonuc* gibi *lexeme*'leri olabilen bir *token*'dır.
- Bazı durumlarda, bir *token*'ın sadece tek bir olası *lexeme*'i vardır. Örneğin, toplama_işlemcisi denilen aritmetik işlemci "+" sembolü için, tek bir olası *lexeme* vardır.
- Boşluk (*space*), ara (*tab*) veya yeni satır karakterleri, *token*'lar arasına yerleştirildiğinde bir programın anlamı değişmez.
- Yandaki örnekte, verilen C deyimi için *lexeme* ve *token*'lar listelenmiştir.

Genel Sentaks tanımlama problemi : Terminoloji

Kısaca

- Bir *cümle (sentence)* herhangi bir alfabede karakterlerden oluşan bir stringdir
- Bir *dil* (*language)* cümlelerden oluşan bir kümedir
- Bir *lexeme* bir dilin en alt seviyedeki sentaktik (syntactic) birimidir (örn., `*`, `sum`, `begin`)
- Bir *simge (token)* lexemelerin bir kategorisidir (örn., tanıtıcı (identifier))

Karakter akışı



Leksikal Analiz (Tarama)

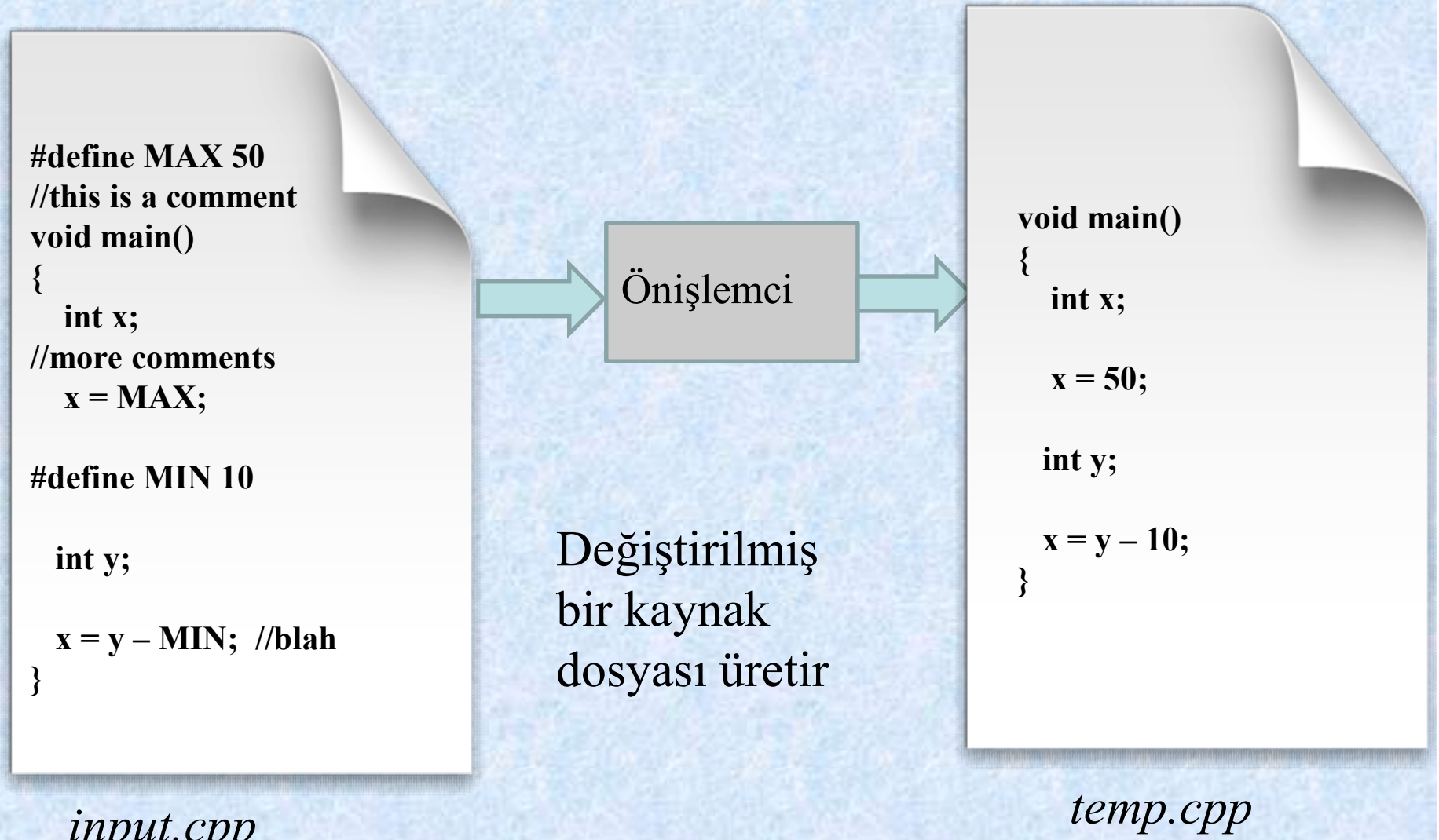
Token akışı

1 (ident) "val"	3 (assign) -	2 (number) 10	4 (times) -	1 (ident) "val"	5 (plus) -	1 (ident) "i"

token numarası

token değeri

Bağımsız Önişlemci



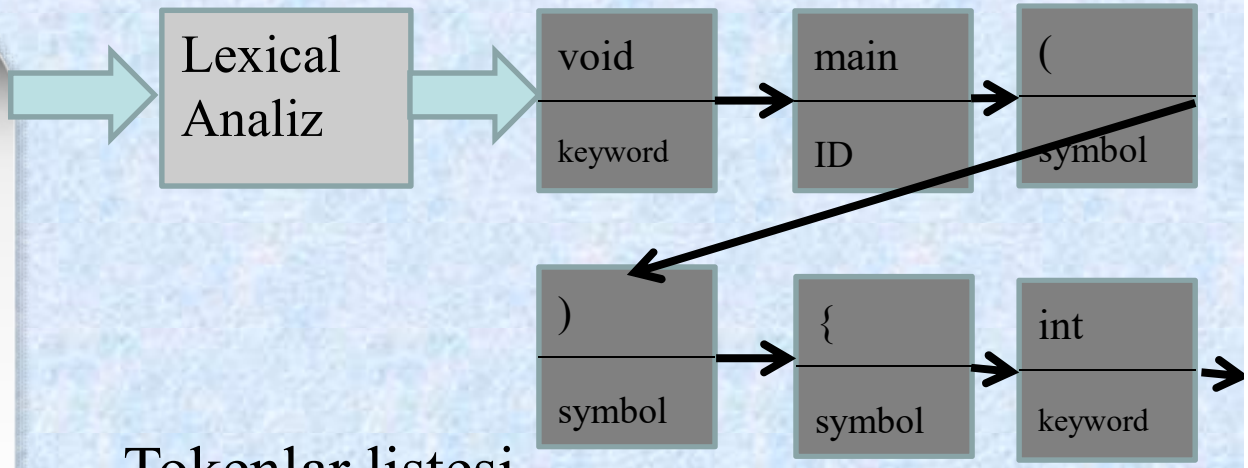
Bağımsız Sözlüksel (Lexical) Analiz

```
void main()
{
   int x;

   x = 50;

  int y;

  x = y – 10;
}
```



Tokenlar listesi üretir

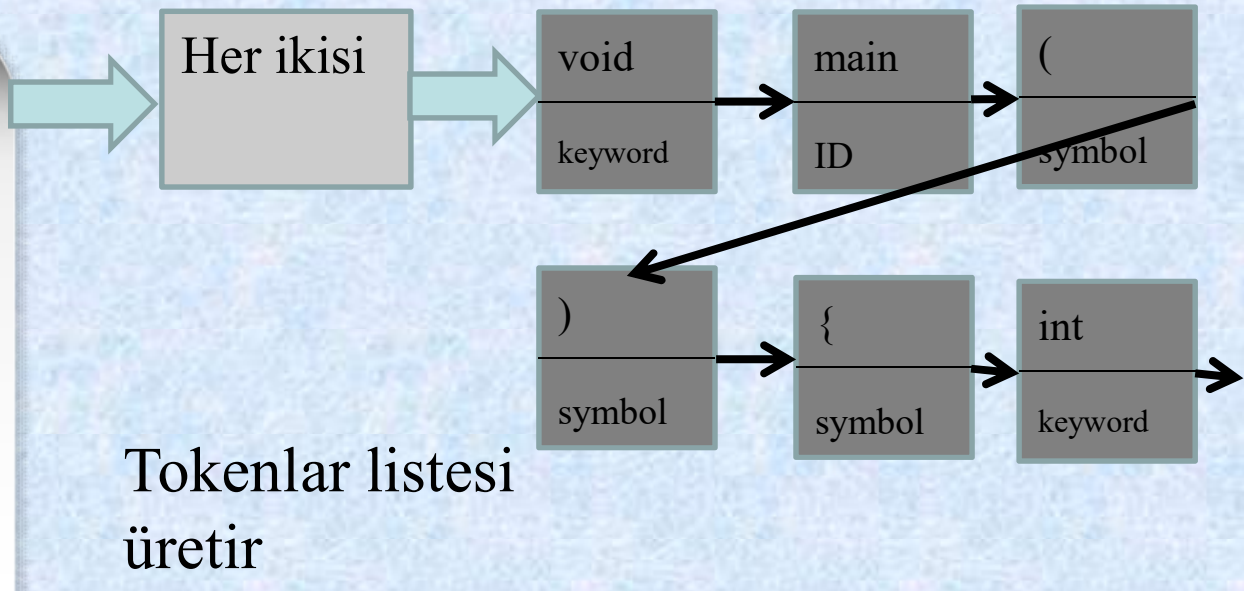
Önişlemci & Sözlüksel Analiz

```
#define MAX 50
//this is a comment
void main()
{
   int x;
//more comments
   x = MAX;

#define MIN 10

  int y;

  x = y – MIN;  //blah
}
```



Tokenlar listesi üretir

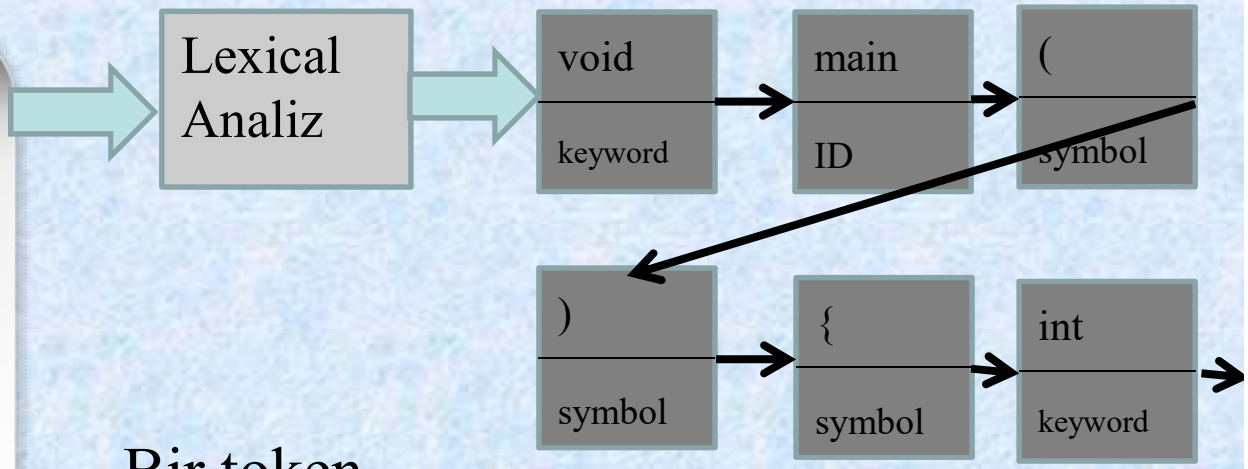
Sözlüksel (Lexical) Analiz

```
void main()
{
   int x;

   x = 50;

  int y;

  x = y – 10;
}
```



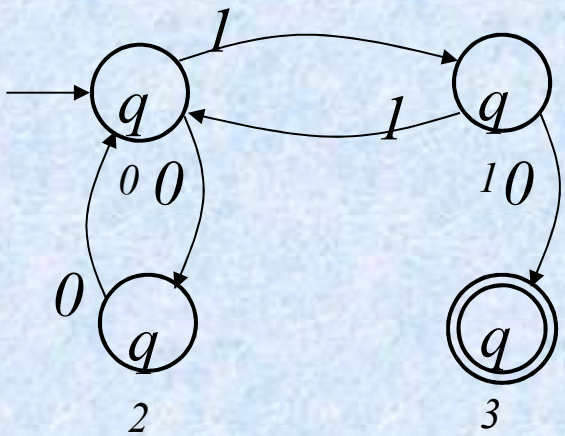
Bir token listesi üretir

Dillerin formal tanımları

- Tanıyıcılar (Recognizers)
  - Bir tanıma aygıtı bir dilin girdi stringlerini okur ve girdi stringinin dile ait olup olmadığına karar verir
  - Örnek: bir derleyicinin sentaks analizi kısmı
- Üreteçler (Generators)
  - Bir dilin cümlelerini üreten aygıttır
  - Belli bir cümlenin sentaksının doğru olup olmadığı, üretecin yapısıyla karşılaştırılarak anlaşılabilir

Dillerin formal tanımları

- Dil Tanıyıcılar
  - Verilen bir programın bir dilde olup olmadığına karar veren bir cihaz
  - Mesela, bir derleyicinin syntax analizcisi sonlu otomat



Sonlu otomatın geçiş diyagramı

$F = (Q, \sum, \delta, q_0, F)$

- Dil üreteciler
  - Bir dilin cümlelerini üretmek için kullanılabilen cihaz
  - Mesela, regular expressions, context-free grammars

$((00)^* 1 (11)^*)^+ 0$

001110 → Kabul

111110 → Kabul

000110 → Red

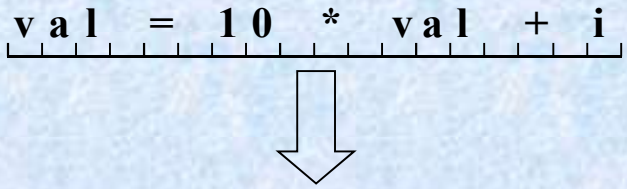
Sentaks tanımlamanın biçimsel metotları

- Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile metadil adı verilir
- Programlama dillerinin sözdizimini anlatmak için BNF (Backus-Naur Form) adlı metadil kullanılacaktır. Öte yandan, anlam tanımlama için böyle bir dil bulunmamaktadır.
- Backus-Naur Form ve İçerik Bağımsız (içerik-bağımsız) (Context-Free) Gramerler
  - Programlama dili sentaksını tanımlamayan en çok bilinen metottur.
- (Genişletilmiş) Extended BNF
  - BNF'un okunabilirliği ve yazılabilirliğini arttırır
- Gramerler ve tanıyıcılar (recognizers)

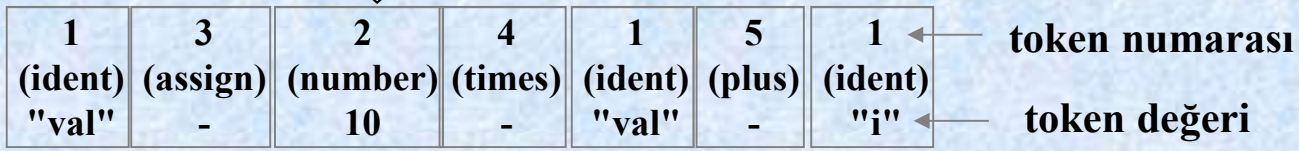
İçerik Bağımsız (Context Free) Gramer

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

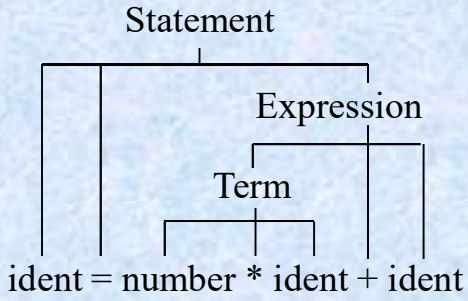
Karakter akışı

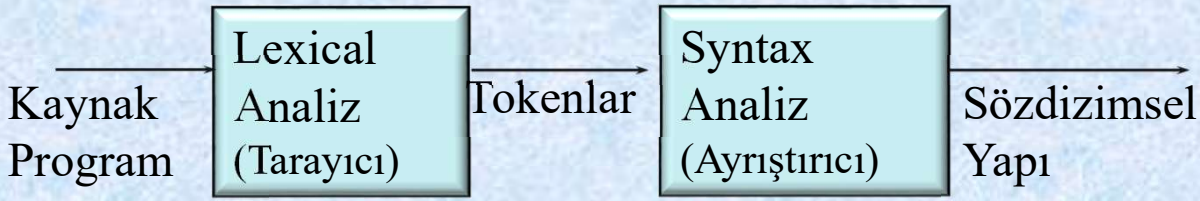


Token akışı

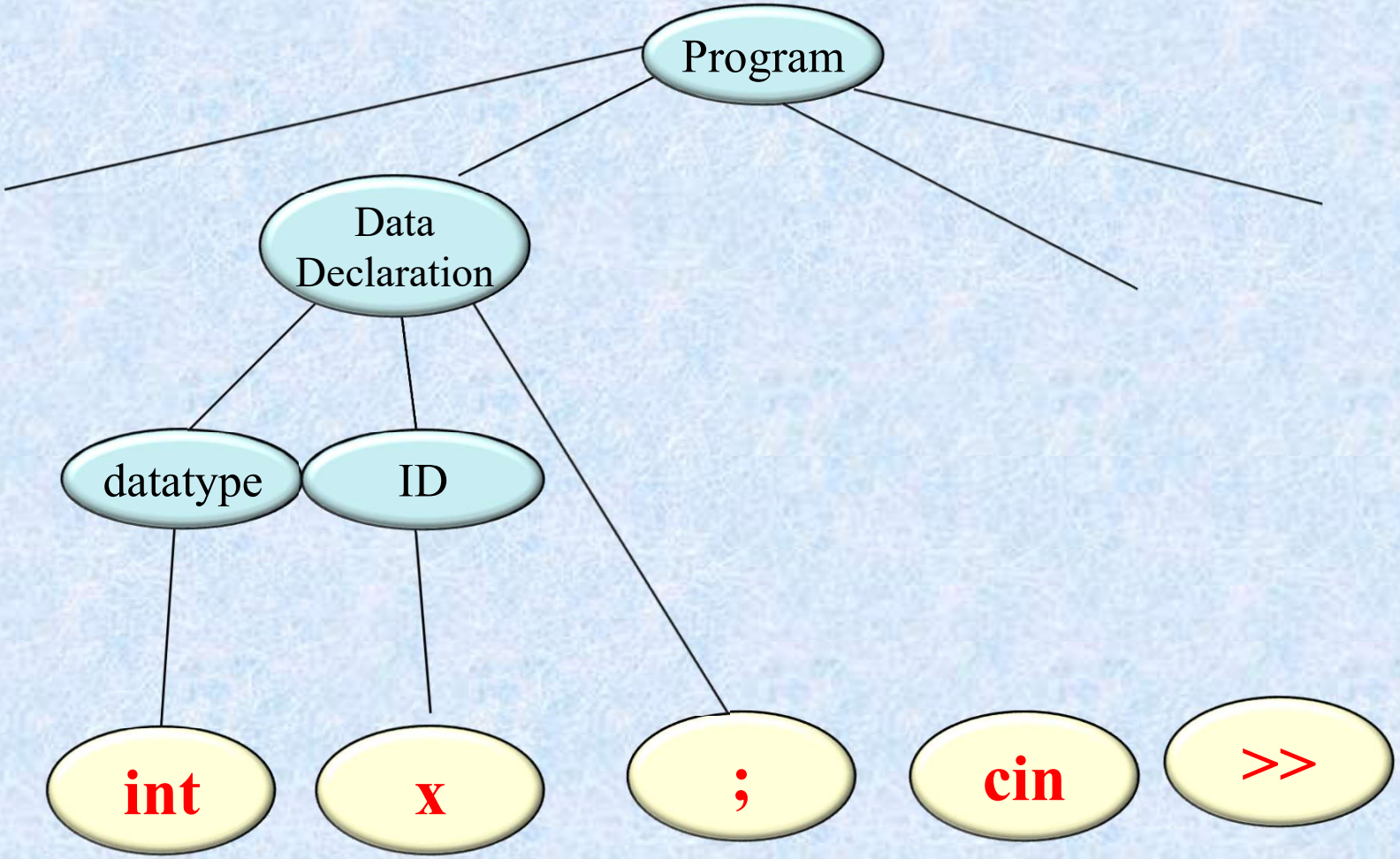


Sentaks ağacı





Ayrıştırma Ağacı



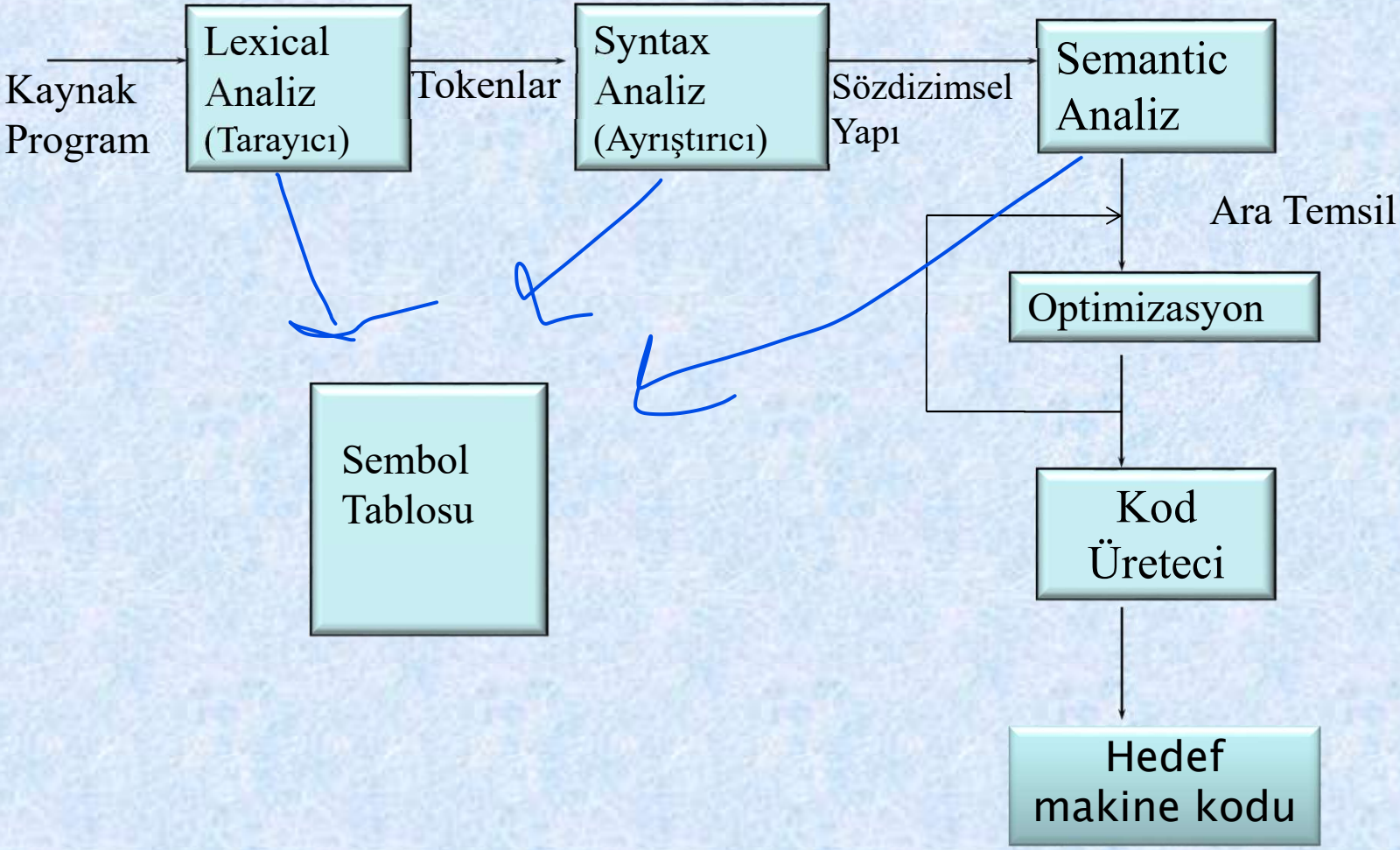
Hatalar

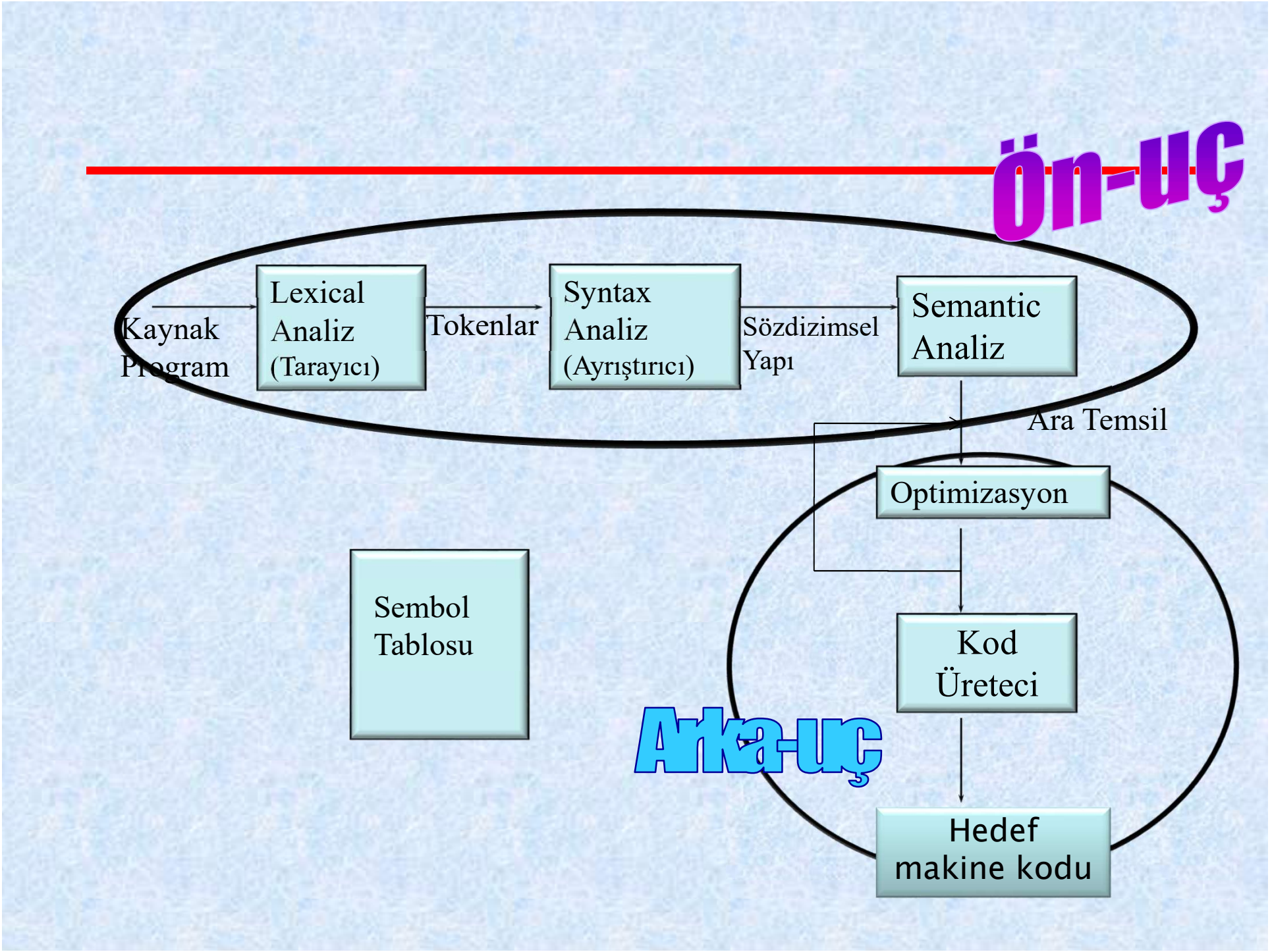
- int x$y;
- int 32xy;
- 45b
- 45ab
- x = x @ y;

Sözlüksel (Lexical) Hatalar / Token Hataları?

- X = ;
- Y = x +;
- Z = [;

Syntax Hataları


Kaynak Program
Lexical Analiz (Tarayıcı)
Tokenlar
Syntax Analiz (Ayrıştırıcı)
Sözdizimsel Yapı
Semantic Analiz
Ara Temsil
Optimizasyon
Sembol Tablosu
Kod Üreteci
Hedef makine kodu


Ön-uç
Kaynak Program
Lexical Analiz (Tarayıcı)
Tokenlar
Syntax Analiz (Ayrıştırıcı)
Sözdizimsel Yapı
Semantic Analiz
Ara Temsil
Optimizasyon
Sembol Tablosu
Kod Üreteci
Arka-uç
Hedef makine kodu

BNF ve İçerik Bağımsız (Context-Free) Gramerler

- İçerik Bağımsız (Context-Free) Gramerler
  - Noam Chomsky tarafından 1950lerin ortalarında geliştirildi
  - Dil üreteçleri (generators), doğal dillerin sentaksını tanımlama amacındaydı
  - İçerik Bağımsız (Context-Free) diller adı verilen bir diller sınıfı tanımlandı
    - Bu dillerin özelliği A → γ şeklinde gösterilmeleridir. Buradaki γ değeri uç birimler (terminals) ve uç birim olmayanlar (nonterminals) olabilmektedir. Bu diller aşağı sürüklemeli otomatlar (push down automata PDA) tarafından kabul edilen dillerdir ve hemen hemen bütün programlama dillerinin temelini oluşturmaktadırlar.

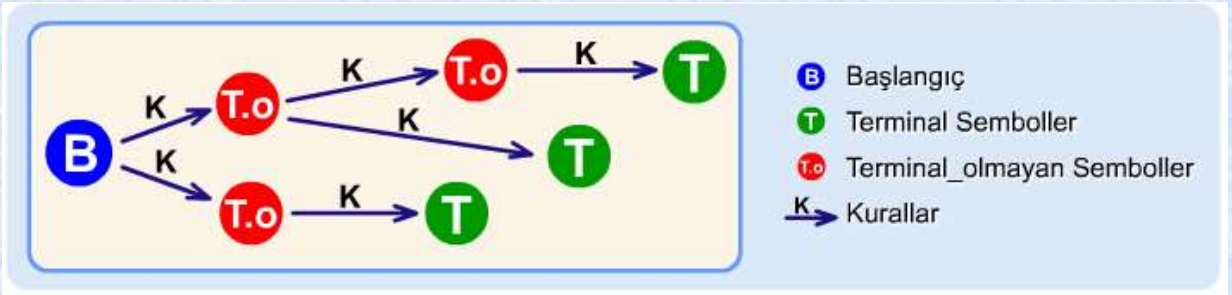
Backus-Naur Form (BNF)

- Backus-Naur Form (1959)
  - John Backus tarafından Algol 58'i belirlemek için icat edildi
  - Bu gösterim şekli, ALGOL60'ıntanımlanması için Peter Naur tarafından biraz değiştirilmiş ve yeni şekli **Backus-Naur (BNF)** formu olarak adlandırılmıştır
  - BNF içerik-bağımsız (context-free) gramerlerin eşdeğeridir
  - BNF başka bir dili tanımlamak için kullanılan bir *metadil*dir
  - BNF'de, soyutlamalar sentaktik (syntactic) yapı sınıflarını temsil etmek için kullanılır--sentaktik değişkenler gibi davranırlar (*nonterminal semboller* adı da verilir)

Backus-Naur Form (BNF) Temelleri

BNF'de açıklanan bir gramer, 4 bölümden oluşur:

1. Terminal Sembolleri (Atomik uç birimler- lexemeler ve simgeler (tokens))
2. Terminal Olmayan Semboller (Sözdizim değişkenleri)
3. Kurallar (Gramer, üretim, Terminal olmayan sembollerin çözümü)
4. Başlangıç Sembolü (Başlangıç terminal olmayan sembol)



Backus-Naur Form (BNF)

- 1. Terminal Semboller: Bir dilde geçerli olan yapıları oluşturmak için birleştirilen daha alt parçalara ayrılamayan (atomik) sembollerdir.
  Örnek: +,*,-,%, if, >=, vb.

- 2. Terminal Olmayan Semboller: Dilin kendisinde bulunmayan, ancak kurallar ile tanımlanan ara tanımları göstermek için kullanılan sembollerdir. BNF'de terminal olmayan semboller "<" ve ">"sembolleri arasında gösterilir ve kurallar ile tanımlanır.
  Örnek: <Statement>, <Expr>, <Type>

Backus-Naur Form (BNF)

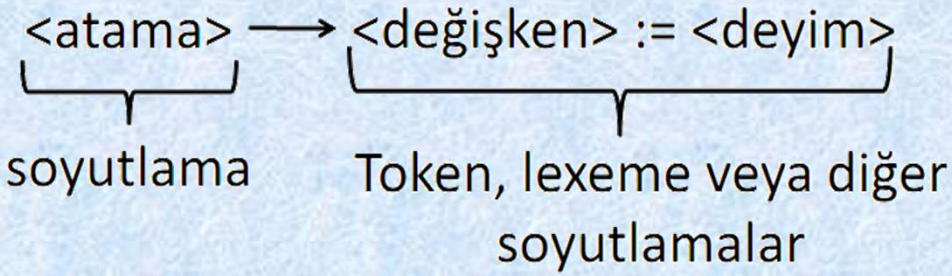
- 3. Kurallar : Bir terminal olmayan sembolün bileşenlerinin tanımlanmasıdır. Her kuralın sol tarafında bir terminal olmayan daha sonra ":=" veya "→" sembolü ve sağ tarafında ise terminal veya terminal olmayanlardan oluşan bir dizi bileşen bulunur.

  Örnek:

  <if_stmt> → **if** <logic_expr> **then** <stmt>

Backus-Naur Form (BNF)

- BNF'deki kurallar, söz dizimsel yapıları göstermek için soyutlamalar (kurallar) olarak düşünülebilir.
- Örnek: Atama deyimi, <atama> soyutlaması ile aşağıdaki gibi belirtilebilir:



- Yukarıdaki soyutlama yapılmadan önce <değişken> ve <deyim> soyutlamalarının daha önceden yapılmış olması gerekmektedir.
- Bir gramer, kuralların boş olmayan sonlu bir kümesidir

Backus-Naur Form (BNF)

- Bir soyutlama (veya kural) için birden çok tanımlama olabilir. Bu durumda bir soyutlama için geçerli olan kurallar “|” ile ayrılır. "|" sembolü veya anlamındadır.
  - Örnek:

```
<exp> → <var>+<var> | <var>-<var>
```

- Bir soyutlama (abstraction) (veya nonterminal sembol) birden fazla RHS’ye sahip olabilir

```
<stmt> → <single_stmt>
       | begin <stmt_list> end
```

Backus-Naur Form (BNF)

Özyinelemeli Kurallar:

- BNF'de bir kural tanımında sol tarafın sağ tarafta yer alması, kuralın özyinelemeli olması olarak açıklanır. Aşağıda görülen <tanımlayıcı_listesi>, özyinelemeli kurallara ve nonterminal sembol için birden çok kural olmasına örnektir.

- <tanımlayıcı_listesi>:=tanımlayıcı | tanımlayıcı, <tanımlayıcı_listesi>
- 4. Başlangıç Sembolü
- BNF'de dilin ana elemanını göstermek için, terminal olmayan sembollerden biri, başlangıç sembolü (amaç sembol) olarak tanımlanır.

Gramerler ve Türetmeler

- 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.
- Bir türetme, başlangıç sembolüyle başlayan ve bir cümleyle (tüm terminal sembolleri) biten kuralların tekrarlamalı bir uygulamasıdır.

Gramerler ve Türetmeler

- Örnek bir gramer :

<program> -> begin <deyim_listesi> end

<deyim_listesi> -> <deyim>
|<deyim>;<deyim_listesi>

<deyim>-> <değişken> :=<ifade>

<ifade> -> <değişken> + <değişken>
|<değişken>

<değişken> -> X | Y | Z

Gramerler ve Türetmeler

- Örnek: Gramer şeklinde görülen dilin, atama görevini gören tek bir deyimi vardır. Bir **program**, *begin* ile başlar, *end* ile biter. Bir **ifade**, ya **tek bir değişkenden** ya da **iki değişken** ve + işlemcisinden oluşabilir. Kullanılabilen **değişken** isimleri *X, Y* veya *Z* dir. Yukarıda verilen gramer aşağıdaki örnek üzerinde açıklanmaktadır:

```
begin
  z:= y+y ;
  x:= z+y
end
```

z, y, x değişkendir.

```
begin
  z:= y+y ;
  x:= z+y
end
```

y+y, z+y ifadedir.

```
begin
  z:= y+y ;
  x:= z+y
end
```

z:= y+y ve
x:= z+y deyimdir.

```
begin
  z:= y+y ;
  x:= z+y
end
```

z:= y+y;
x:= z+y
deyim listesidir.

```
begin
  z:= y+y ;
  x:= z+y
end
```

Tamamı bir programda yer alan bir bloğu oluşturur.

Gramerler ve Türetmeler

- Bu türetmedeki başlangıç sembolü *<program>* dır.
- Her cümle, bir önceki cümledeki terminal_olmayanlardan birinin tanımının yerleştirilmesiyle türetilir.
- Bu türetmede, yeni bir satırda tanımı yapılan terminal_olmayan, her zaman bir önceki satırda yer alan en soldaki terminal_olmayandır.
- Bu sıra ile oluşturulan türetmelere **sola_dayalı** türetme adı verilir. Türetme işlemi, sadece terminallerden veya *lexeme* lardan oluşan bir cümle oluşturulana kadar devam eder.

- Bir sola_dayalı_türetmede, terminal_olmayanları yerleştirmek için farklı sağ taraf kuralları seçerek, dildeki farklı cümleler oluşturulabilir. Bir türetme, sola_dayalı türetmeye ek olarak, **sağa_dayalı** olarak veya **ne sağa_dayalı ne de sola_dayalı** olarak oluşturulabilir.

Gramerler ve Türetmeler

- Bu dildeki bir programın türetilmesi aşağıdaki örnek türetme üzerinde görülmektedir.

Örnek

<program> -> begin <deyim_listesi> end

-> begin <deyim>; end

-> begin <değişken> := <ifade>; end

->begin X := <ifade>; end

->begin X:=<değişken>+<değişken>; end

->begin X := Y + <değişken>; end

->begin X:=Y+Z; end

<program> -> begin <deyim_listesi> end

<deyim_listesi> -> <deyim>
|<deyim>;<deyim_listesi>

<deyim>-> <değişken> :=<ifade>

<ifade> -> <değişken> + <değişken>
|<değişken>

<değişken> -> X | Y | Z

Bir Türetme (derivation) Örneği

```
<program> → <stmts>
<stmts> → <stmt> | <stmt> ; <stmts>
<stmt> → <var> = <expr>
<expr> → <term> + <term> | <term> - <term>
<term> → <var> | const
<var> → a | b | c | d
```

<program> => <stmts> => <stmt>
=> <var> = <expr>
=> a =<expr>
=> a = <term> + <term>
=> a = <var> + <term>
=> a = b + <term>
=> a = b + const

Türetme (Derivation)

- Bir türetmede yar alan bütün sembol stringleri cümlesel biçimdedir (sentential form)
- Bir cümle (sentence) sadece terminal semboller içeren cümlesel bir biçimdir
- Bir ensol türetme (leftmost derivation), içindeki her bir cümlesel biçimdeki ensol nonterminalin genişletilmiş olmadığı türetmedir
- Bir türetme ensol (leftmost) veya ensağ (rightmost) dan her ikisi de olmayabilir

Gramer ve türetme örneği

Gramer

a=b*(a+c)

<assign> → <id>=<expr>

<id> → a | b | c

<expr> → <id> + <expr>

|<id > * <expr>

|(<expr>)

| id

Gramer ve türetme örneği

Sola dayalı türetme

a=b*(a+c)

<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)

Gramer

a=b*(a+c)

<assign> → <id>=<expr>

<id> → a | b | c

<expr> → <id> + <expr>

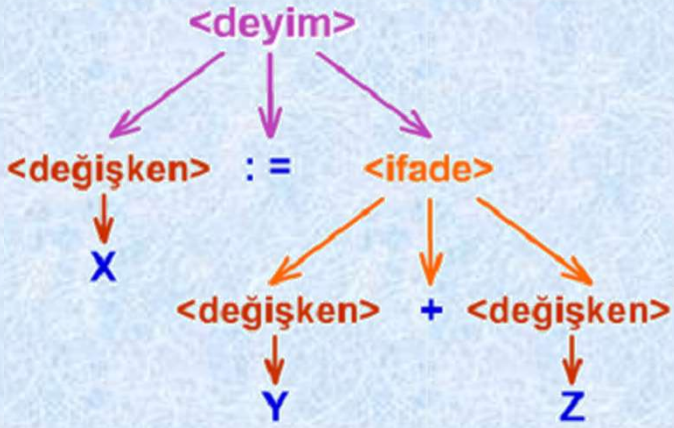
|<id > * <expr>

|(<expr>)

| id

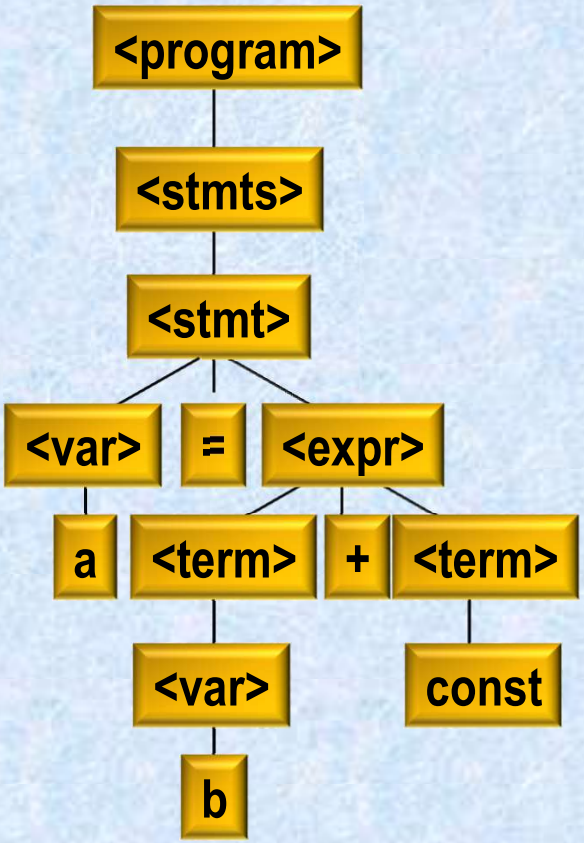
Ayrıştırma Ağacı (Parse Tree)

- Gramerler, tanımladıkları dilin cümlelerinin hiyerarşik sözdizimsel yapısını tarif edebilirler. Bu hiyerarşik yapılara **ayrıştırma (parse) ağaçları** denir. Bir ayrıştırma ağacının en aşağıdaki düğümlerinde terminal semboller 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, "X := Y + Z", deyiminin yapısını göstermektedir.



Ayrıştırma Ağacı (Parse Tree)

- Bir türetmenin (derivation) hiyerarşik gösterimi



Gramer ve türetme örneği

Sola dayalı türetme

a=b*(a+c)

<assign> => <id>=<expr>

=>a=<expr>

=>a=<id>*<expr>

=>a=b*<expr>

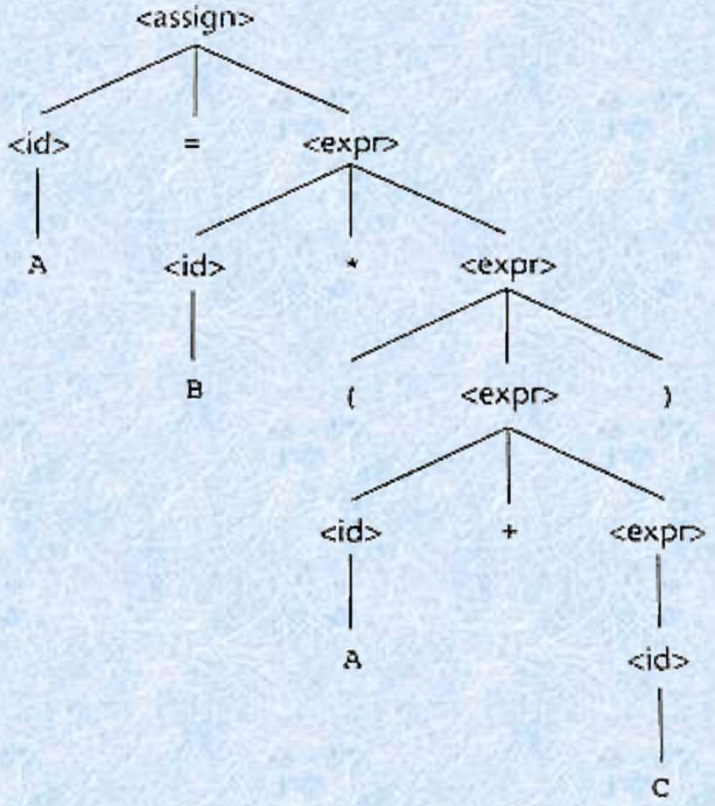
=>a=b*(<expr>)

=>a=b*(<id>+<expr>)

=>a=b*(a+<expr>)

=>a=b*(a+<id>)

=>a=b*(a+c)



Gramerlerde Belirsizlik (Ambiguity)

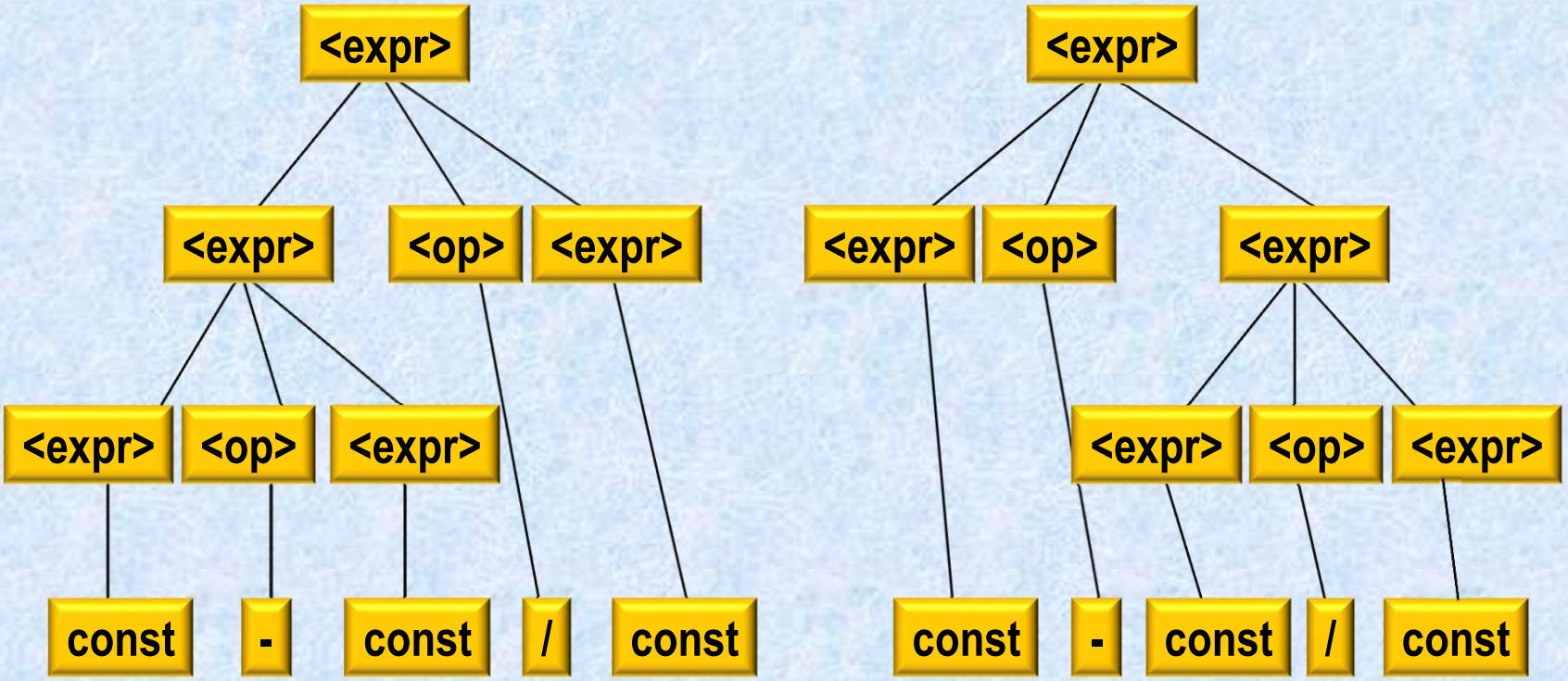
- Bir gramer ancak ve ancak iki veya daha fazla farklı ayrıştırma ağacı olan bir cümlesel biçim (sentential form) üretiyorsa *belirsiz*dir

Bir Belirsiz Deyim Grameri

<expr> → <expr> <op> <expr> | const

<op> → / | -

const – const / const



Bir Belirsiz Olmayan (Unambiguous) Deyim Grameri

- Eğer ayrıştırma ağacını operatörlerin öncelik seviyelerini göstermek için kullanırsak, belirsizlik olmaz.

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

const – const / cont

Türetme:

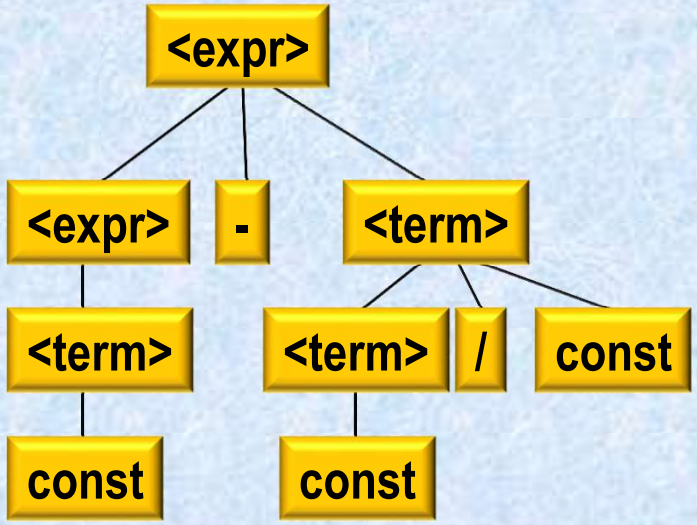
<expr> ➔ <expr> – <term>

➔ <term> – <term>

➔ const – <term>

➔ const – <term> / const

➔ const – const / const

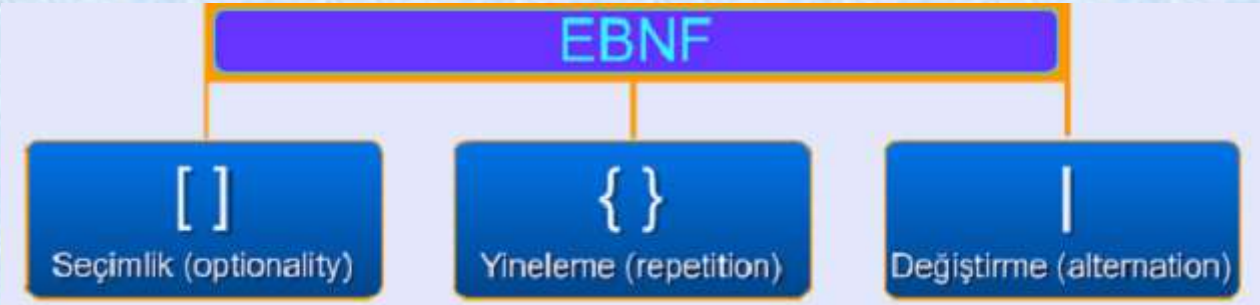


Genişletilmiş BNF (Extended BNF())

- BNF'nin okunabilirliğini ve yazılabilirliğini artırmak amacıyla, BNF'e bazı eklemeler yapılmış ve yenilenmiş BNF sürümlerine genişletilmiş BNF veya kısaca EBNF adı verilmiştir.

  EBNF' in özellikleri :

- EBNF'te Seçimlik (optionality), Yineleme (repetition) ve Değiştirme (alternation) olmak üzere üç özellik yer almaktadır:



Genişletilmiş BNF (Extended BNF())

- Seçimlik (*optionality*) [ ]
- Bir kuralın sağ tarafında, isteğe bağlı olarak yer alabilecek bir bölümü belirtmek için [ ] kullanımı eklenmiştir. [ ] içindeki bölüm, bir kural tanımında hiç yer almayabilir veya bir kez bulunabilir. Örneğin C'deki *if* deyimi aşağıdaki şekilde gösterilebilir:

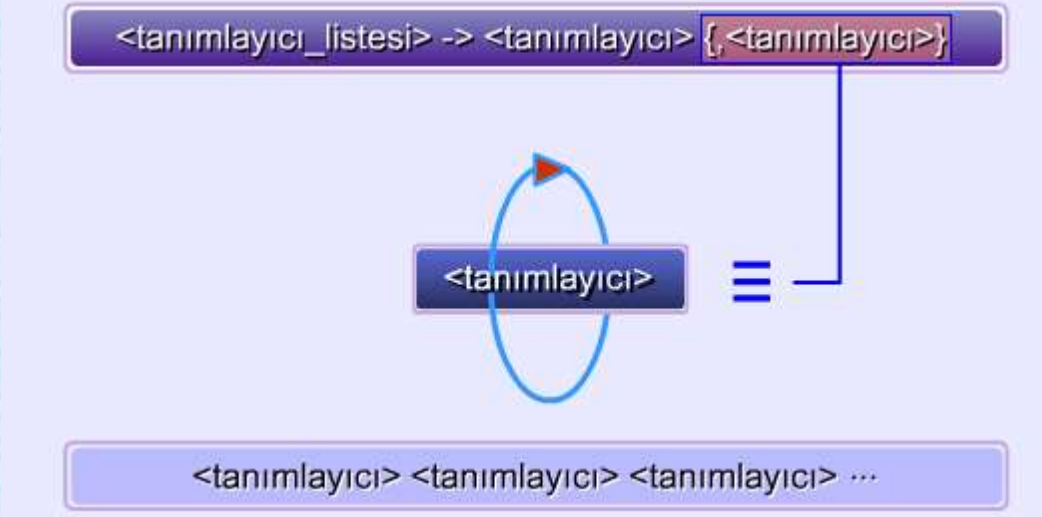
<seçimlik_deyim> -> If (<mantıksal>) <deyim> [else <deyim>];

- [ ] kullanılmadığı durumda, bu *if* deyiminin aşağıda gösterildiği gibi iki kural ile açıklanması gereklidir:

EBNF	<seçimlik_deyim> -> **If** (<mantıksal>) <deyim> [**else** <deyim>];
BNF	<seçimlik_deyim> -> **If** (<mantıksal>) <deyim>
	<seçimlik_deyim> -> **If** (<mantıksal>) <deyim> **else** <deyim>;

Genişletilmiş BNF (Extended BNF())

- Yineleme *(repetition)* { }
- Bir kuralın sağ tarafında, istenilen sayıda yinelenebilecek veya hiç yer almayabilecek bir bölümü göstermek için { } kullanımı eklenmiştir. EBNF'deki yineleme sembolü ile, BNF'de iki kural olarak gösterilen tanımlamalar, tek kural ile ifade edilebilmektedir.
  - Örnek: dogal sayi ::= sifir haric sayi , { sayi } ; Bu durumda, *1*, *2*, ...,*10* ,..., *12345*,... değerleri doğru ifadelerdir.



Genişletilmiş BNF (Extended BNF())

- Değiştirme *(alternation)* |
- Bir grup içinden tek bir eleman seçilmesi gerektiği zaman seçenekler, parantezler içinde birbirlerinden "*veya*" işlemcisi "|" ile ayrılarak yazılabilir. Aşağıda, Pascal'daki *for* deyimi için gerekli kural gösterilmektedir. Bu yapıyı BNF'te göstermek için iki kural gerekli iken, değiştirme sembolü ile EBNF gösteriminde tek kural yeterli olmaktadır.

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

ÖZET: Genişletilmiş BNF

- Seçimlik kısımlar köşeli parantez içine yerleştirilir ([ ])

  `<proc_call> -> ident [(<expr_list>)]`

- RHS lerin (sağ-taraf) alternatif değiştirme kısımları parantezler içine yerleştirilir ve dikey çizgilerle ayrılır

  `<term> → <term> (+|-) const`

- Yinelemeler (Repetitions) (0 veya daha fazla) süslü parantez ({ }) içine yerleştirilir

  `<ident> → letter {letter|digit}` brace

ÖZET: Genişletilmiş BNF

- EBNF'de yer alan [ ],{ } ve | sembolleri, gösterimi kısaltmaya yarayan metasembollerdir.

<amaç>:= [x] y {z}

y	x seçilmedi, z yinelenmedi.
xy	x seçildi, z yinelenmedi.
yz	x seçilmedi, z bir defa yinelendi.
xyz	x seçildi, z bir defa yinelendi.
yzz	x seçilmedi, z iki defa yinelendi.
xyzz	x seçildi, z iki defa yinelendi.
yzzzz	x seçilmedi, z dört defa yinelendi.

BNF ve EBNF

- BNF

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

- EBNF

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

Özellik (Attribute) Gramerleri

- İçerik-bağımsız gramerler (CFGs) bir programlama dilinin bütün sentaksını tanımlayamazlar
- Ayrıştırma ağaçlarıyla birlikte bazı semantik bilgiyi taşıması için CFG'lere eklemeler (herşeyi gramerde veremeyiz, parse ağacı büyür)
  - Tip uyumluluğu
  - Bazı dillerde değişkenlerin kullanılmadan önce tanımlanması zorunluluğu
- Özellik (attribute) gramerlerinin (AGs) birincil değerleri :
  - Statik semantik belirtimi
  - Derleyici tasarımı (statik semantik kontrolü)

Özellik (Attribute) Gramerleri: Tanım

- Bir özellik grameri G = (S, N, T, P) aşağıdaki eklemelerle birlikte bir içerik-bağımsız gramerdir :
  - Her bir *x* gramer sembolü için özellik değerlerinden oluşan bir *A(x)* kümesi vardır
  - Her kural, içindeki nonterminallerin belirli özelliklerini (attributes) tanımlayan bir fonksiyonlar kümesine sahiptir
  - Her kural, özellik tutarlılığını kontrol etmek için karşılaştırma belirtimlerinden (predicate) oluşan (boş olabilir) bir kümeye sahiptir

Özellik Gramerleri: Tanım

- $X_0 \rightarrow X_1 \ldots X_n$ bir kural olsun
- $S(X_0) = f(A(X_1), \ldots, A(X_n))$ biçimindeki fonksiyonlar *sentezlenmiş özellikleri* tanımlar
- $I(X_j) = f(A(X_0), \ldots, A(X_n))$, $i <= j <= n$ için, şeklindeki fonksiyonlar *miras alınmış özellikleri* tanımlar
- Başlangıçta, yapraklarda *yerleşik özellikler* vardır

Özellik Gramerleri: Örnek

- Sentaks

```
<assign> -> <var> = <expr>
<expr> -> <var> + <var> | <var>
<var> A | B | C
```

- `actual_type`: `<var>` ve `<expr>` ile sentezlenmiştir
- `expected_type`: `<expr>` ile miras bırakılmıştır

Özellik Gramerleri: Örnek

- Sentaks kuralı: **<expr> → <var>[1] + <var>[2]**
  Semantik kurallar:

<expr>.actual_type ← <var>[1].actual_type

Karşılaştırma belirtimi (Predicate):

<var>[1].actual_type == <var>[2].actual_type

<expr>.expected_type == <expr>.actual_type

- Sentaks kuralı: **<var> → id**
  Semantik kuralı:

<var>.actual_type ← lookup (<var>.string)

Özellik Gramerleri: Örnek

- Özellik değerleri nasıl hesaplanır?
  - Eğer bütün özellikler miras alınmışsa, ağaç yukarıdan-aşağıya (top-down order) şekilde düzenlenir
  - Eğer özellikler sentezlenmişse, ağaç aşağıdan-yukarıya (bottom-up order) şekilde düzenlenir.
  - Çoğu kez, bu iki çeşit özelliğin her ikisi de kullanılır, ve aşağıdan-yukarıya ve yukarıdan-aşağıya düzenlerin kombinasyonu kullanılmalıdır.

Özellik Gramerleri: Örnek

<expr>.expected_type ← ebeveyninden miras almıştır

<var>[1].actual_type ← lookup (A)
<var>[2].actual_type ← lookup (B)
<var>[1].actual_type =? <var>[2].actual_type

<expr>.actual_type ← <var>[1].actual_type
<expr>.actual_type =? <expr>.expected_type

Özellik Gramerleri – Bir Örnek

- Nitelikler: *actual_type* (sentezlenen nitelik), *expected_type* (miras kalan nitelik)

1. Sentaks kuralı: <assign> → <var> = <expr>
   Semantik kuralı: <expr>.*expected_type* ← <var>.*actual_type*
2. Sentaks kuralı: <expr> → <var>[2] + <var>[3]
   Semantik kuralı: <expr>.*actual_type* ←
   if (<var>[2].*actual_type* = int) and
   (<var>[3].*actual_type* = int) then int
   else real
   endif
   Predicate: <expr>.*actual_type* = <expr>.*expected_type*
3. Sentaks kuralı: <expr> → <var>
   Semantik kuralı: <expr>.*actual_type* ← <var>.*actual_type*
   Predicate: <expr>.*actual_type* = <expr>.*expected_type*
4. Sentaks kuralı: <var> → A | B | C
   Semantik kuralı: <var>.*actual_type* ← look-up(<var>.*string*)

Nitelik Değerlerini Hesaplama – Nitelikleri Değerlendirme

Cümle: A = A + B

1. <var>.*actual_type* ← look-up(A) (Kural 4)
2. <expr>.*expected_type* ← <var>.*actual_type* (Kural 1)
3. <var>[2].*actual_type* ← look-up(A) (Kural 4)
   <var>[3].*actual_type* ← look-up(B) (Kural 4)
4. <expr>.*actual_type* ←
   int ya da real (Kural 2)
5. <expr>.*expected_type* =
   <expr>.*actual_type*

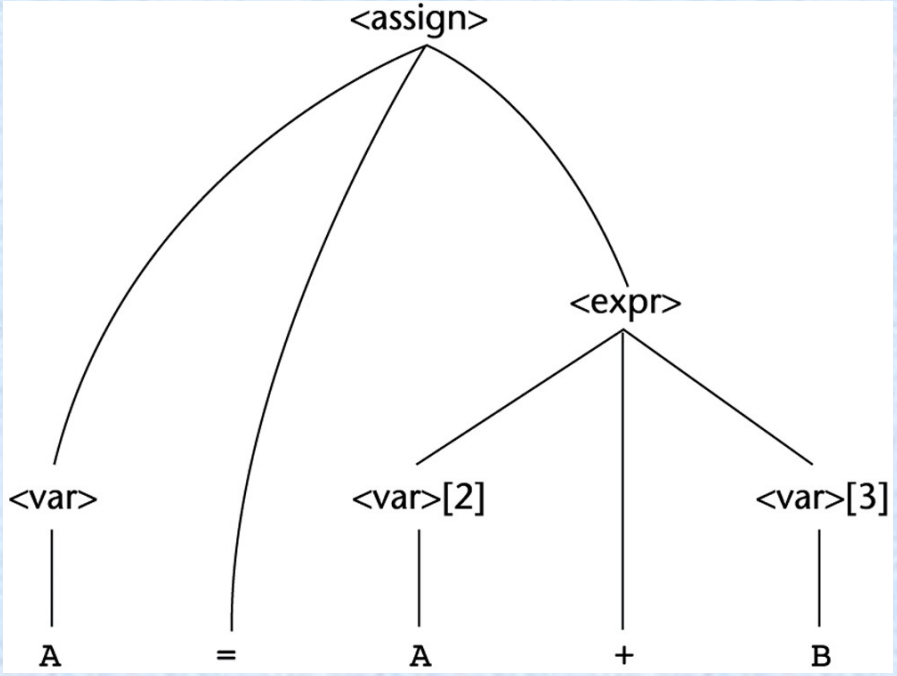
TRUE ya da FALSE'tur (Kural 2)

Gramer:

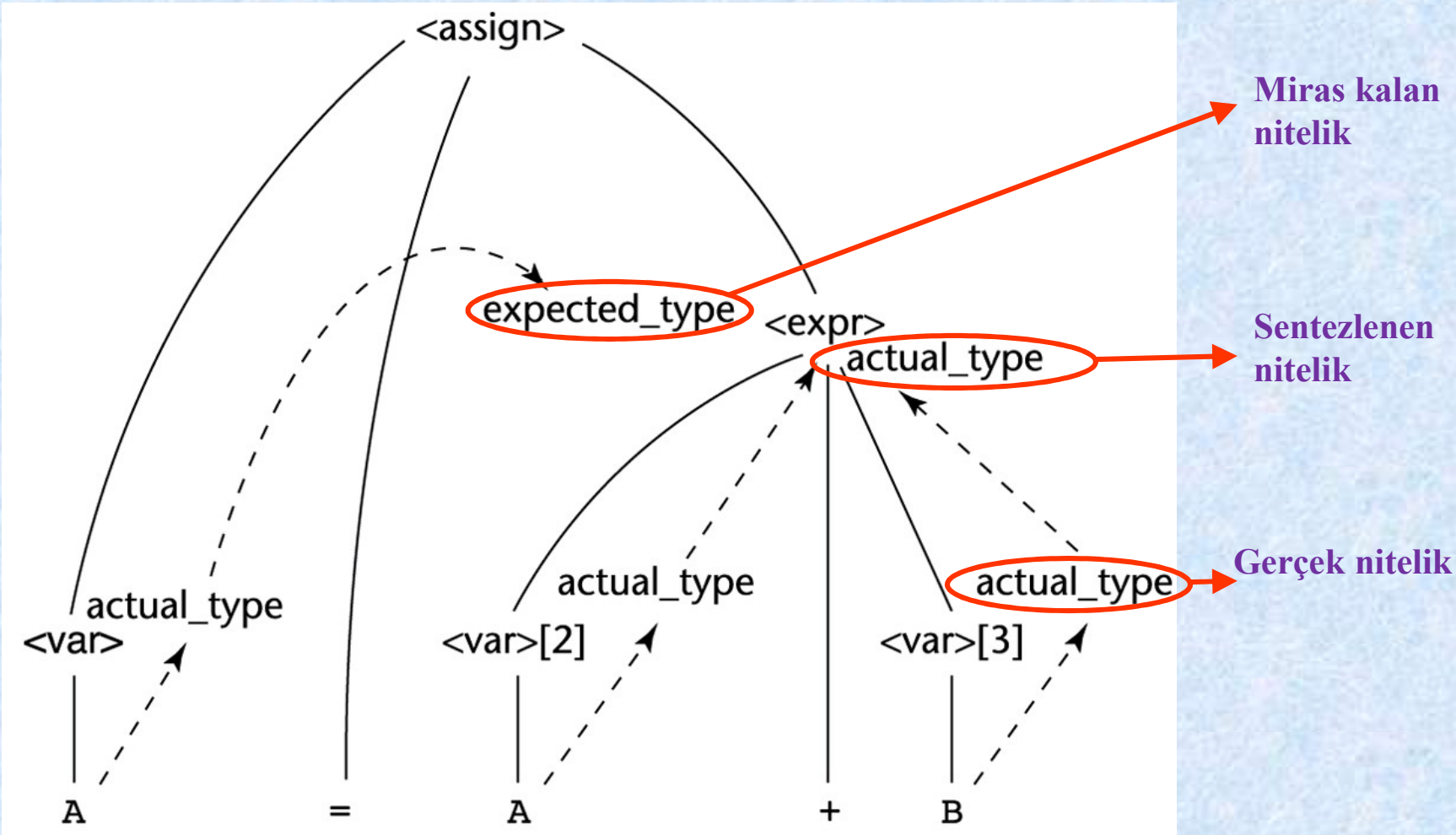
<assign> → <var> = <expr>

<expr> → <var> + <var> | <var>

<var> → A | B | C



Nitelik Değerlerini Hesaplama – Ayrıştırma Ağacında Nitelik Akışı



Nitelik Değerlerini Hesaplama –
Tam bağlanmış nitelik ağacı