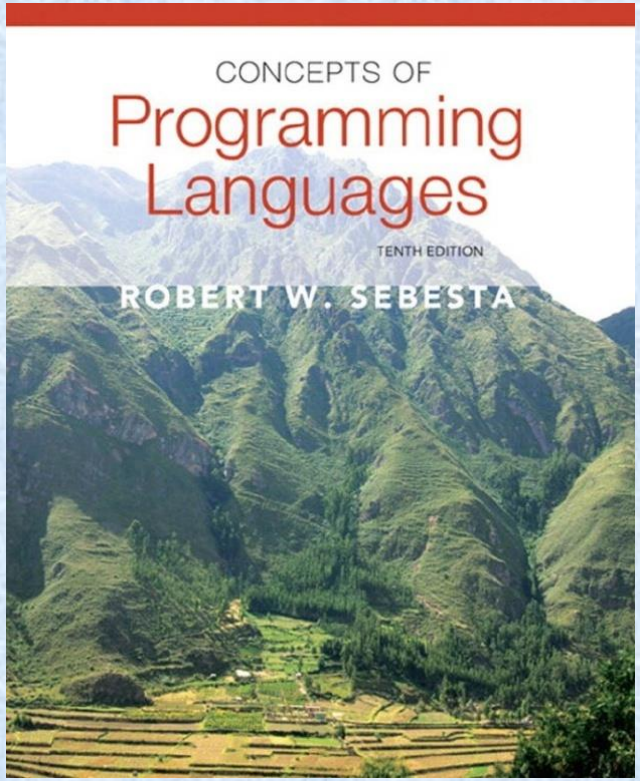
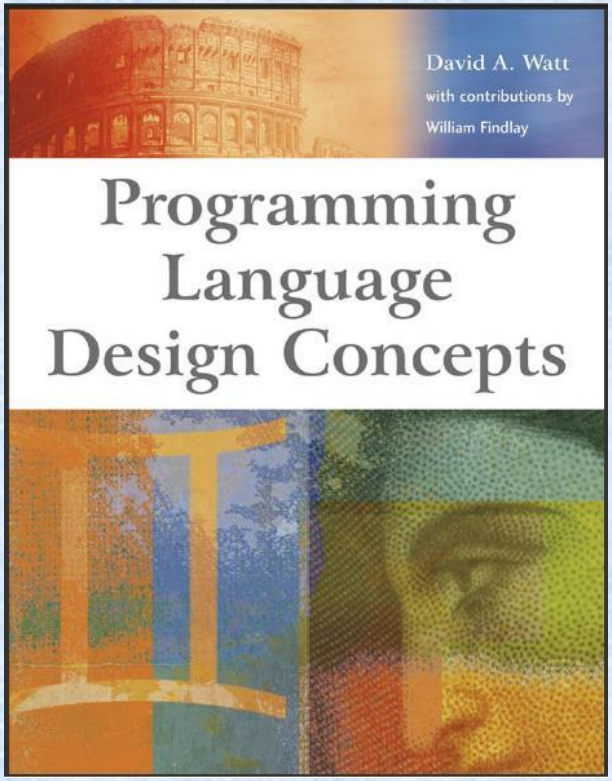
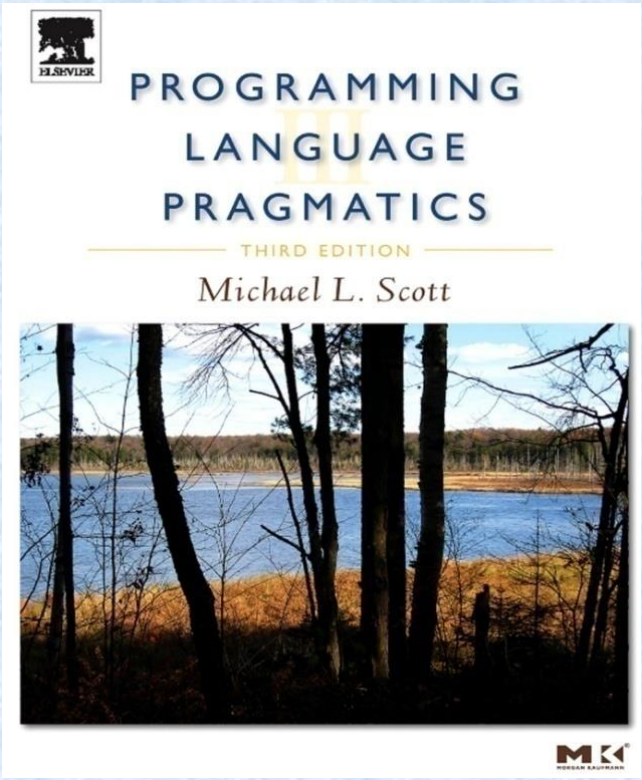
Bölüm 3: Sentaks ve Semantiği Tanımlama







Bölüm 3 Konuları

1. Giriş
2. Genel Sentaks Tanımlama Problemi
3. Sentaks Tanımlamanın Biçimsel Metotları
4. Özellik (Attribute) Gramerleri
5. Programların Anlamlarını Açıklamak: Dinamik Semantik

3.1 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 tanımı 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;}
```

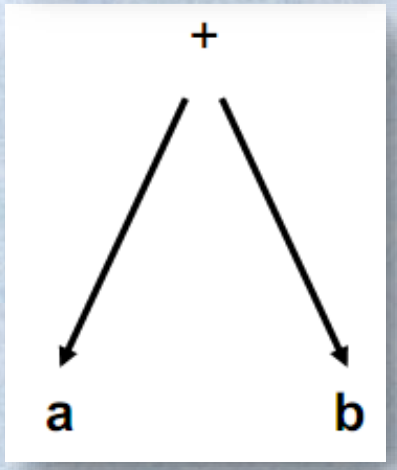
```
while (i<10)
{ a[i]= i++;}
```

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

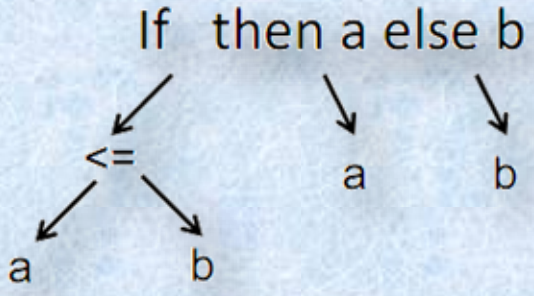


Soyut Sözdizim

Soyut Sözdizim Ağaçları

- Bir ifadedeki işlemci/işlenen yapısını gösteren ağaçlara **soyut sözdizim ağaçları** adı verilir.
- Soyut sözdizim ağaçları, bir 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



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

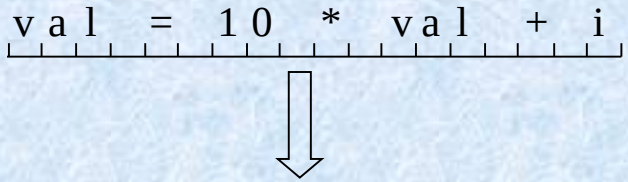
- Bir programlama dilinin metinsel sözdizimi, *token*'lar ile tanımlanır. Örneğin bir tanımlayıcı; ***toplam*** veya ***sonuc*** gibi *lexeme*'leri olabilen bir *token*'dır.
- Bazı durumlarda, bir *token*'ın sadece tek bir olası *lexeme*'i vardır. Örneğin, toplama_işlemcisi denilen aritmetik işlemci "+" 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.

3.2 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)

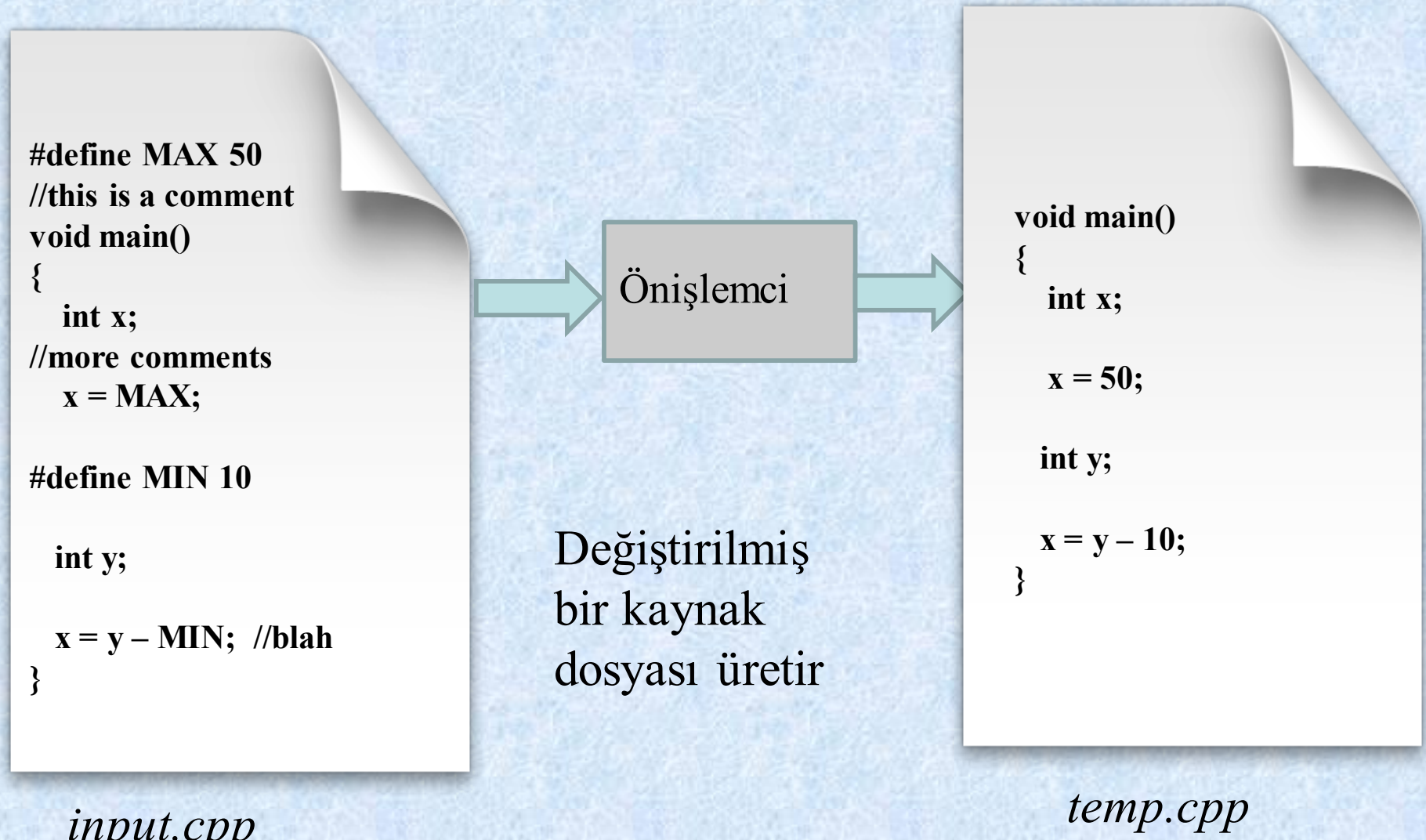
Token akışı

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

token numarası

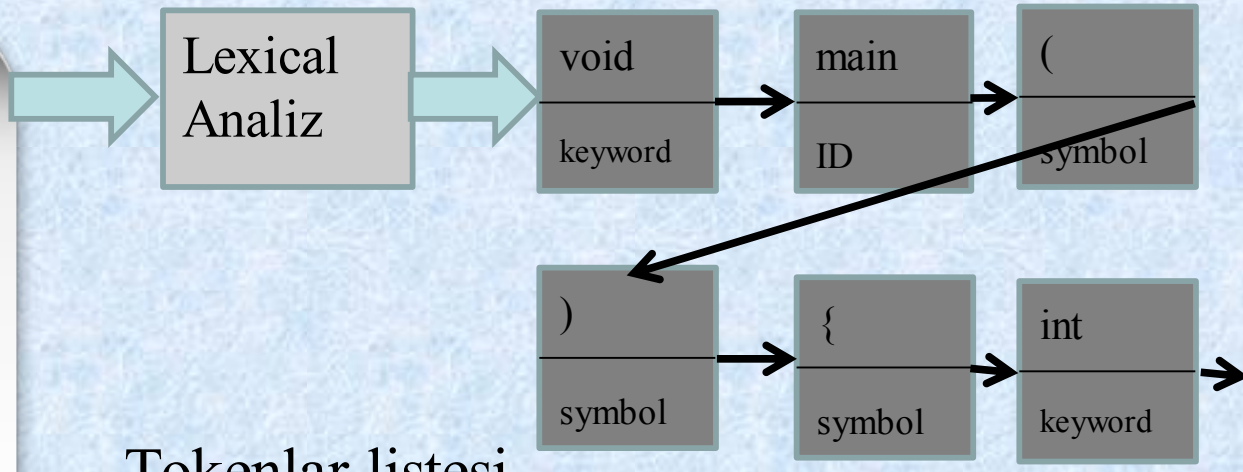
token değeri

Bağımsız Önişlemci



Bağımsız Sözlüksel (Lexical) Analiz

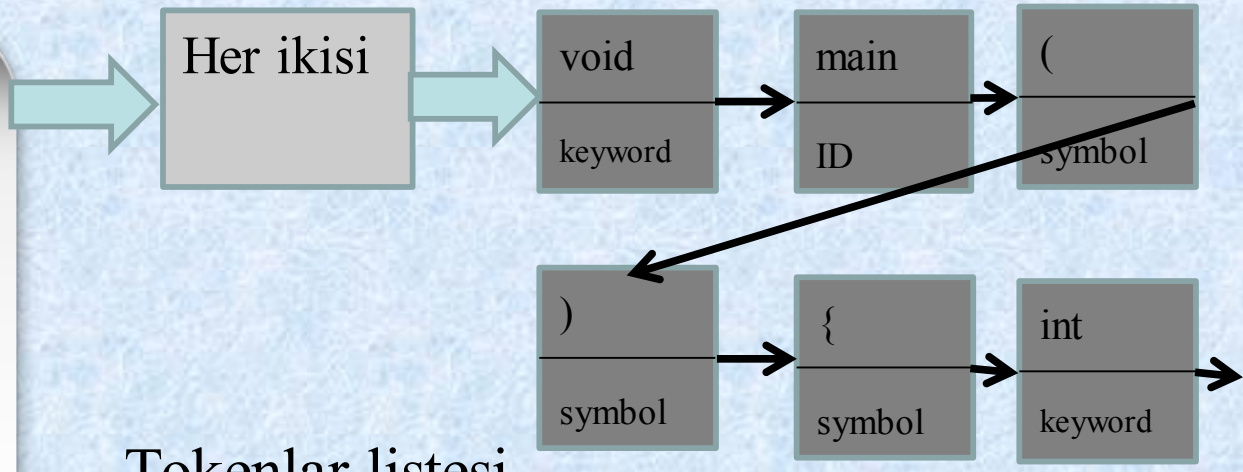
```
void main()
{
   int x;

   x = 50;

  int y;

  x = y – 10;
}
```



Tokenlar listesi üretir

Önişlemci & Sözlüksel Analiz

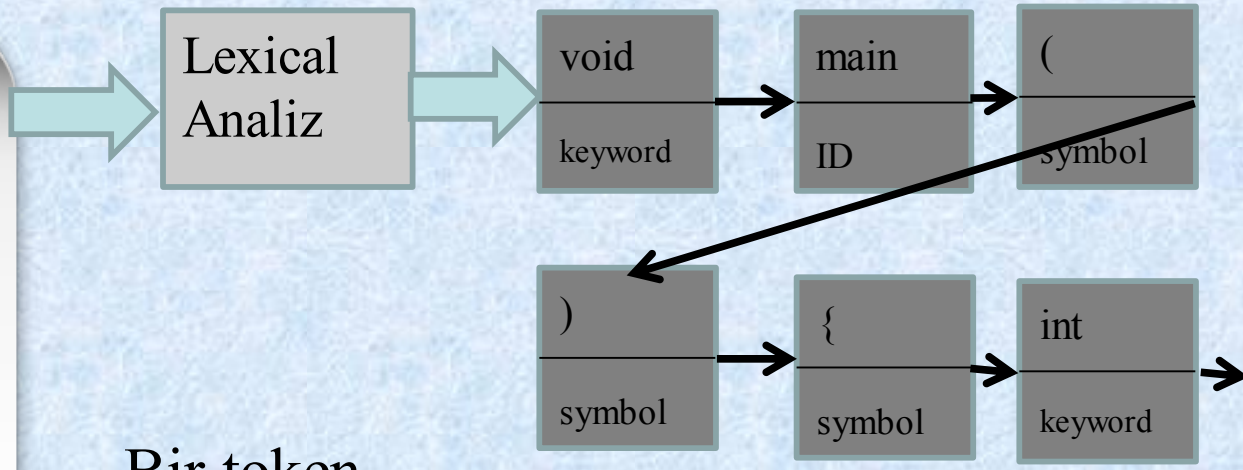
```
#define MAX 50
//this is a comment
void main()
{
   int x;
//more comments
   x = MAX;

#define MIN 10

  int y;

  x = y – MIN;  //blah
}
```



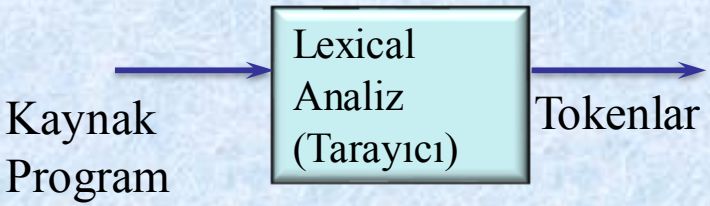
Tokenlar listesi üretir

Sözlüksel (Lexical) Analiz

```
void main()
{
   int x;

   x = 50;

  int y;

  x = y – 10;
}
```



Bir token listesi üretir



Boşluklar? Sorun olur mu?

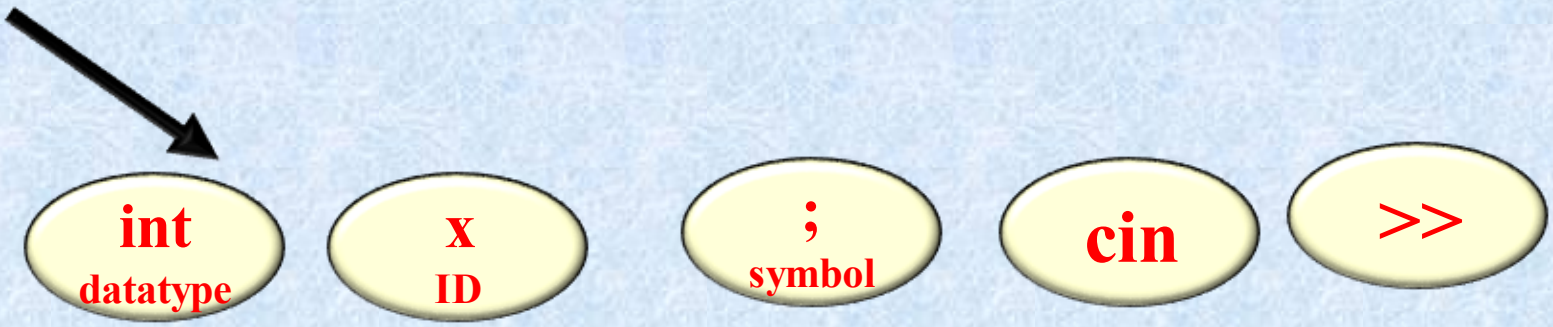
```
int x;
cin >> x;
if(x>5)
   cout << "Hello";
else
   cout << "BOO";
```

```
int   x   ;
cin  >>   x  ;
if  ( x  >  5  )
   cout  <<  "Hello" ;
else
   cout  <<  "BOO"   ;
```

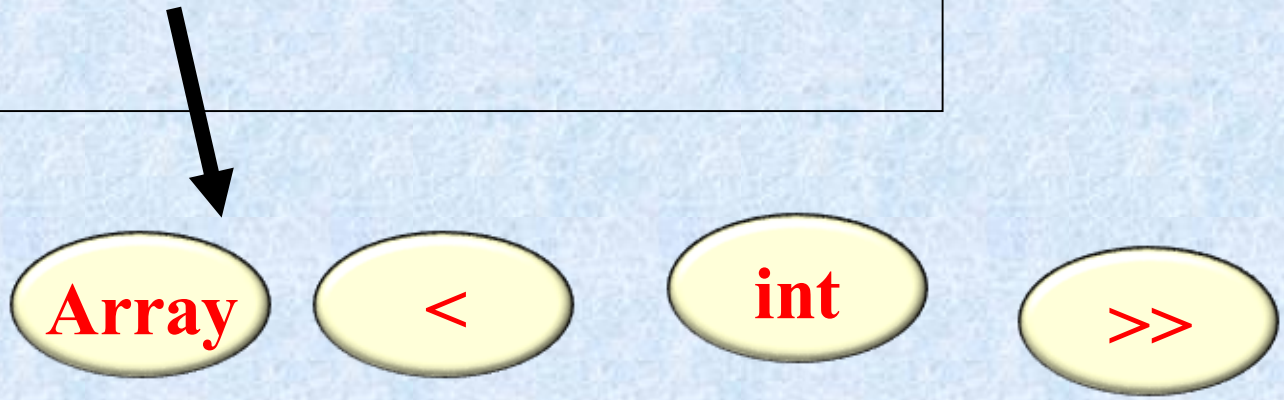
```
int x;
cin >> x;
if(x>5)
    cout << "Hello";
else
    cout << "BOO";
```

Tokenlar = Bir programda anlamlı öğeler

Değer/Tip ikilisi



```
Array<Array<int>> someArray;
```



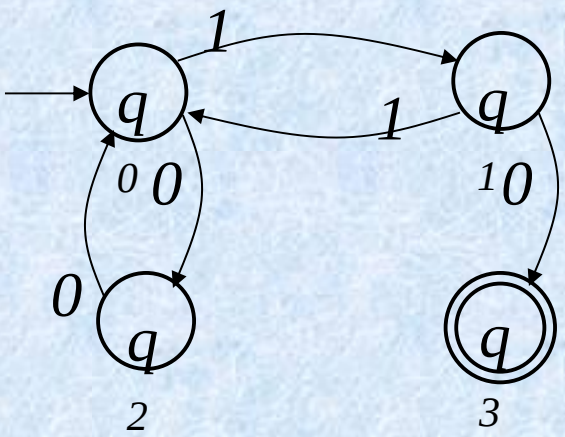
Array<Array<int> > someArray;

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
  - Bölüm 4'te daha detaylı anlatılacak
- **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*

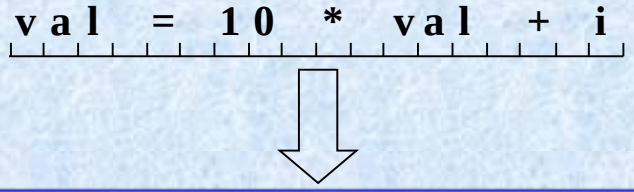
3.3 Sentaks tanımlamanın biçimsel metotları

- 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)

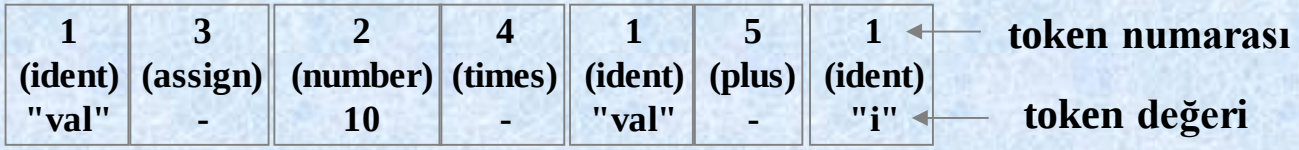
İçerik Bağımsız (Context Free) Gramer

- **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.

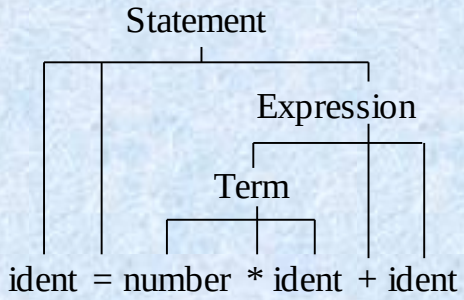
Karakter akışı

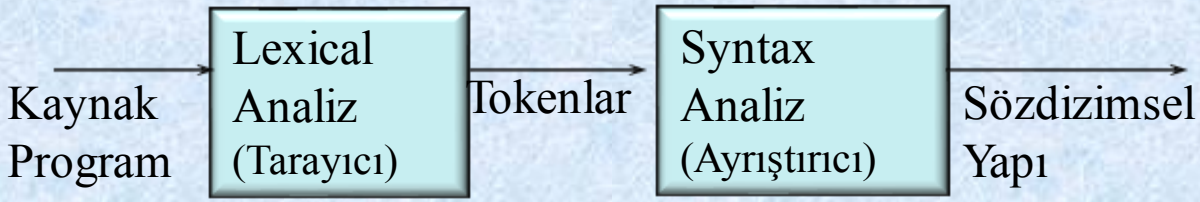


Token akışı

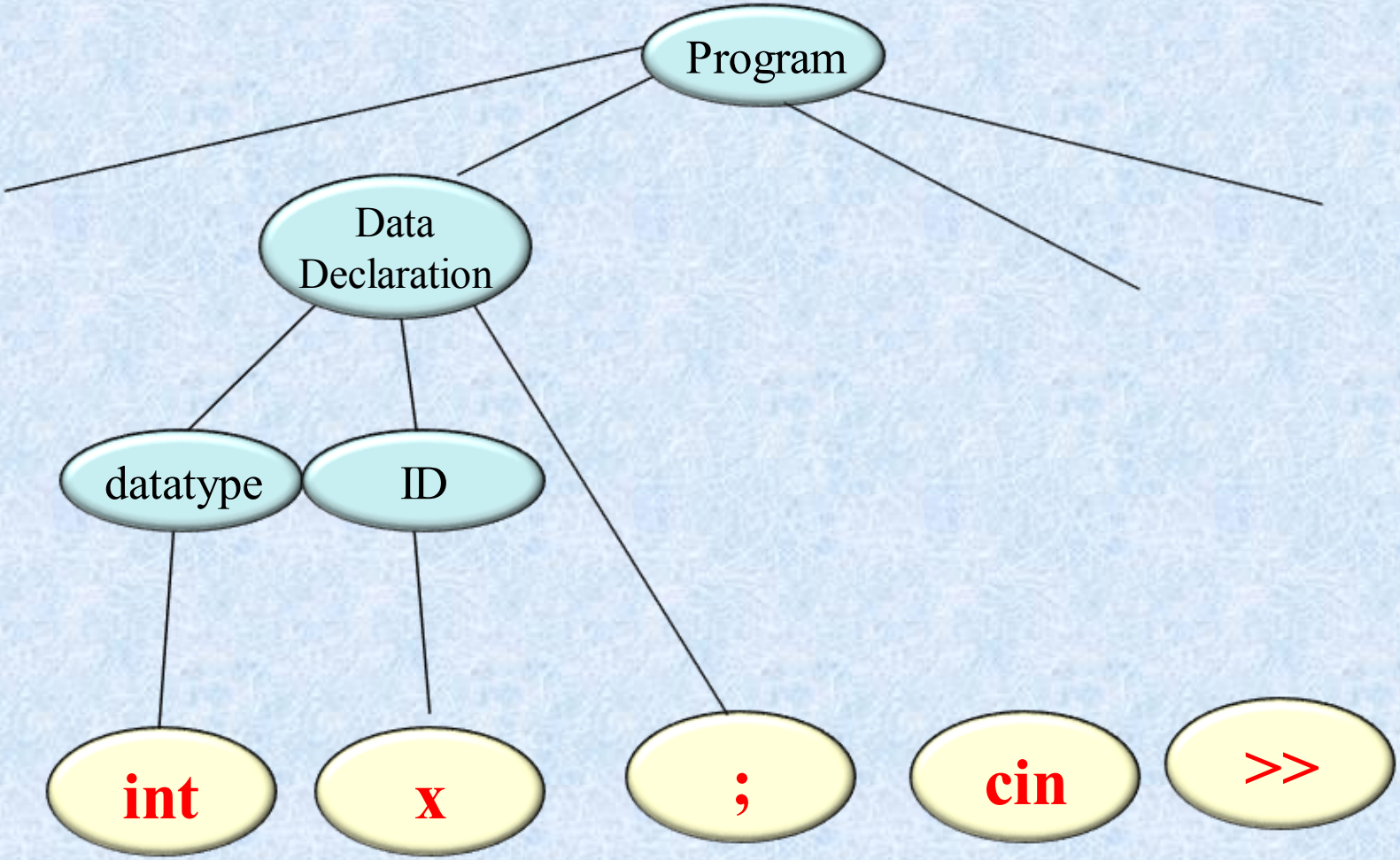


Sentaks ağacı





Ayrıştırma Ağacı



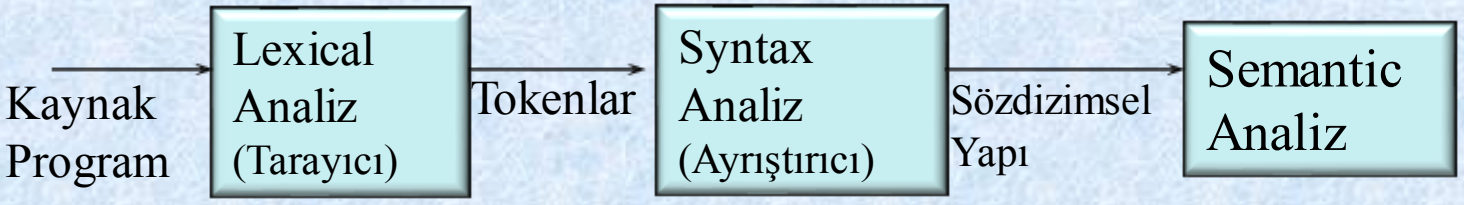
Hatalar

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

Sözlüksel (Lexical) Hatalar / Token Hataları?

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

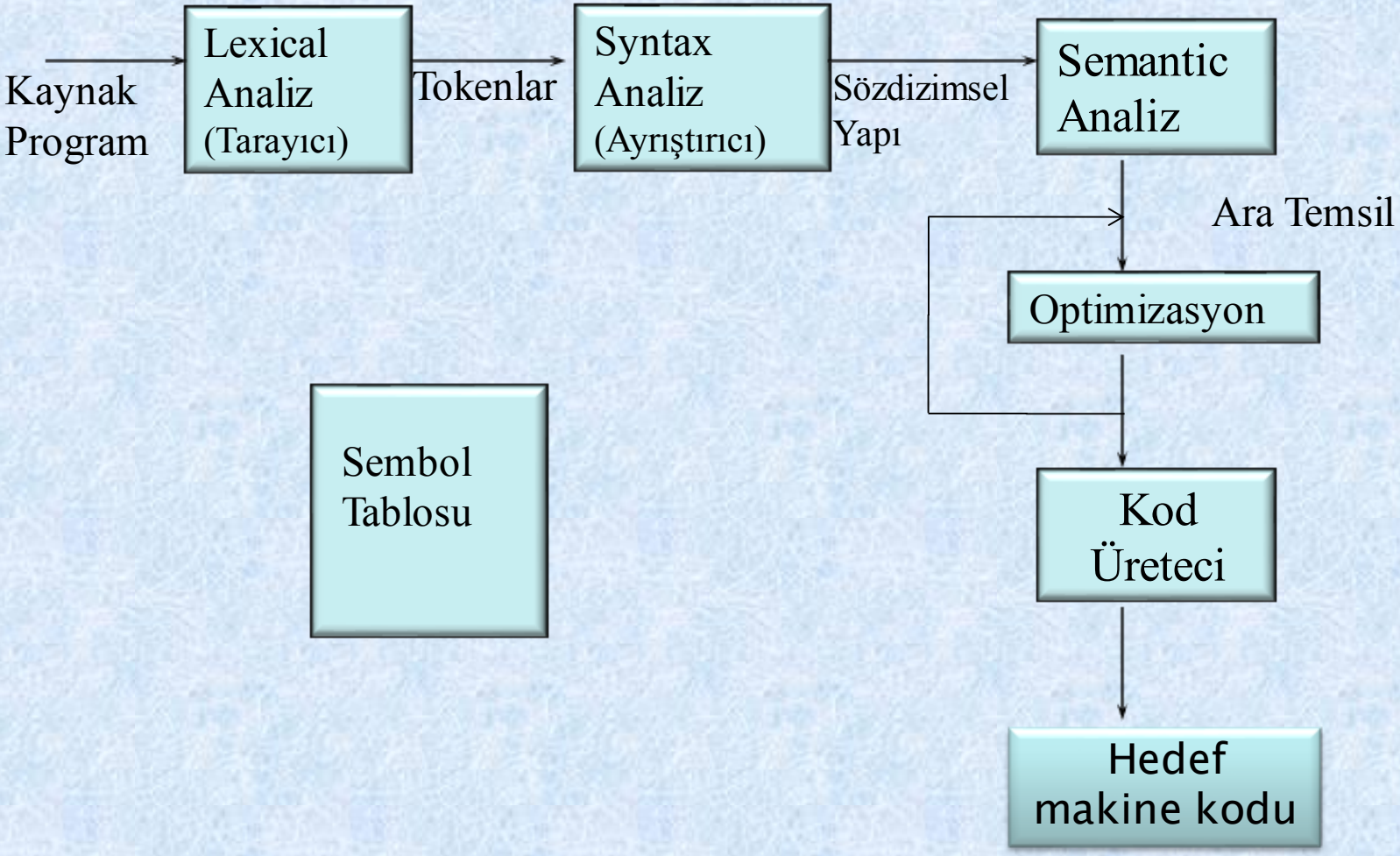
Syntax Hataları

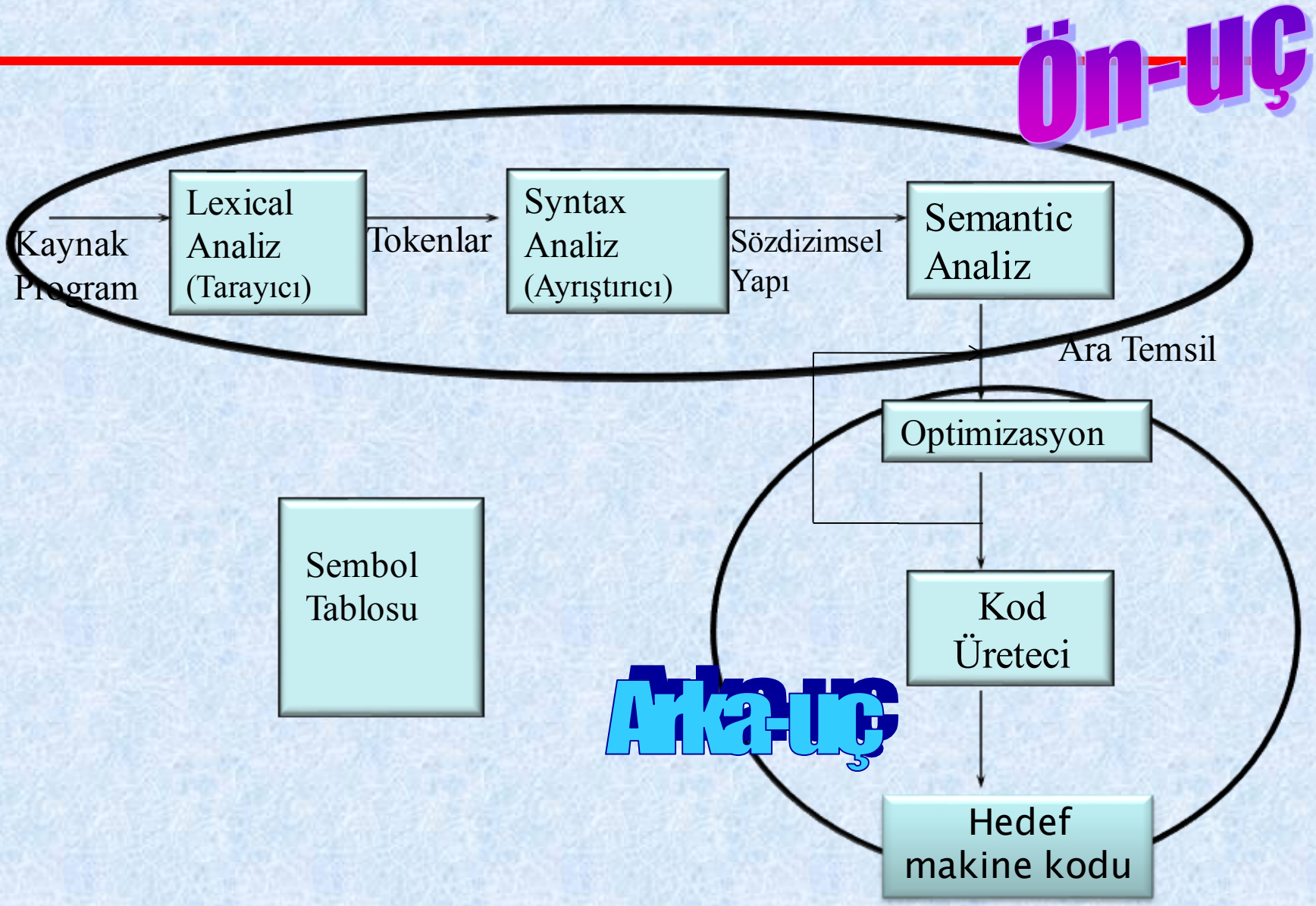


Ayrıştırma Ağacı

Sembol Tablosu

```
int x;
cin >> x;
if(x>5)
   x = "SHERRY";
else
   cout << "BOO";
```


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.

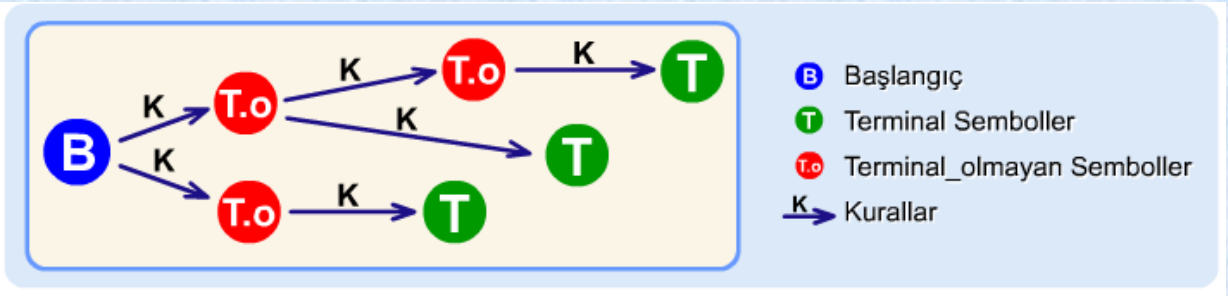
Backus-Naur Form (BNF)

- 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)

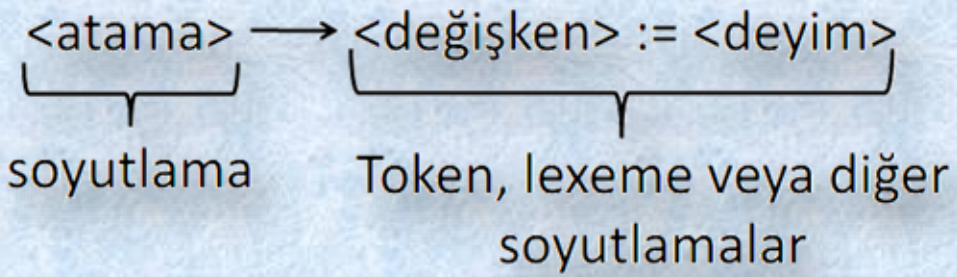
- **3. Kurallar :** Bir terminal olmayan sembolün bileşenlerinin tanımlanmasıdır. Her kuralın sol tarafında bir terminal olmayan daha sonra “:=” veya “→” sembolü ve sağ tarafında ise terminal veya terminal olmayanlardan oluşan bir dizi bileşen bulunur.

  Örnek:

```
<ident list> → identifier | identifer,
  <ident list>
<if_stmt> → if <logic_expr> then <stmt>
```

Backus-Naur Form (BNF)

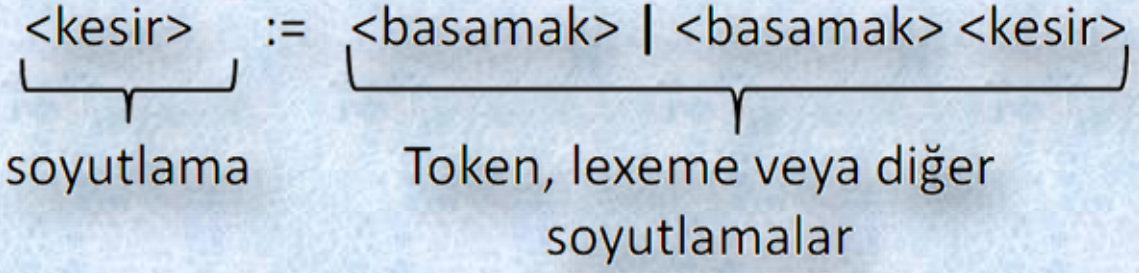
- BNF'deki kurallar, söz dizimsel yapıları göstermek için soyutlamalar (kurallar) olarak düşünülebilir.
- Örnek: Atama deyimi, <atama> soyutlaması ile aşağıdaki gibi belirtilebilir:



- 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:



- 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

Bir Gramer Örneği

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

Bir Türetme (derivation) Örneği

```
<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>

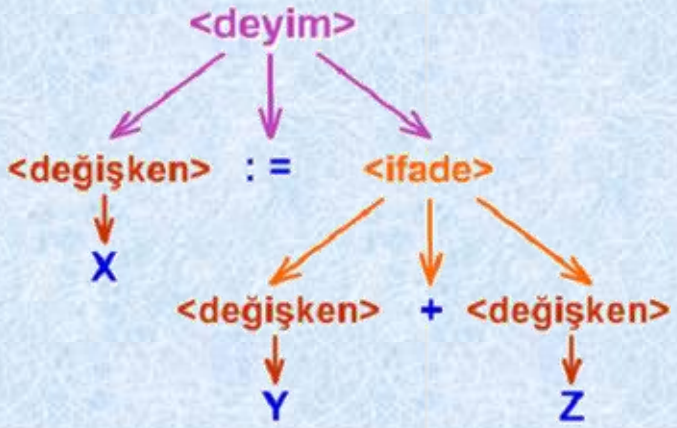
|<id > * <expr>

|(<expr>)

| id

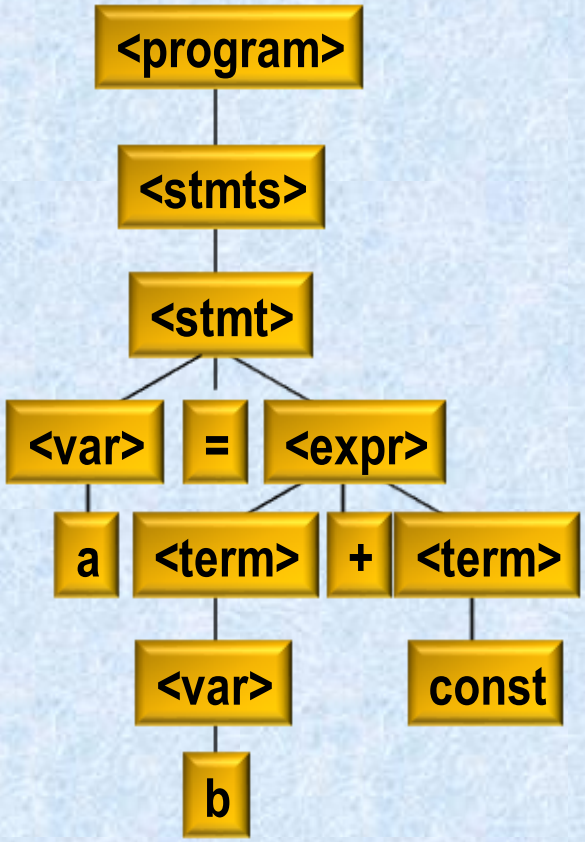
Ayrıştırma Ağacı (Parse Tree)

- 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>

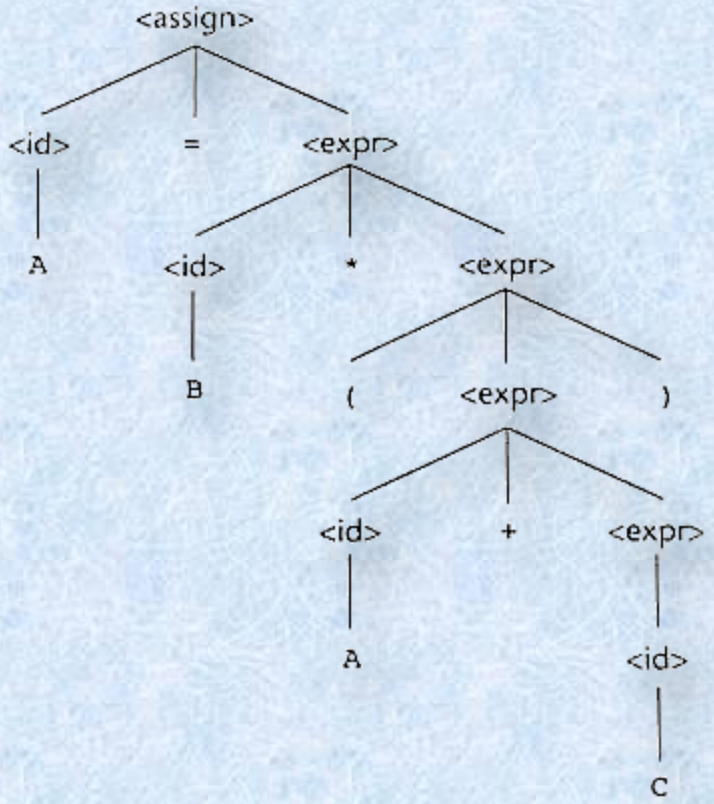
=>a=b*(<expr>)

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

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

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

=>a=b*(a+c)



Gramerlerde Belirsizlik (Ambiguity)

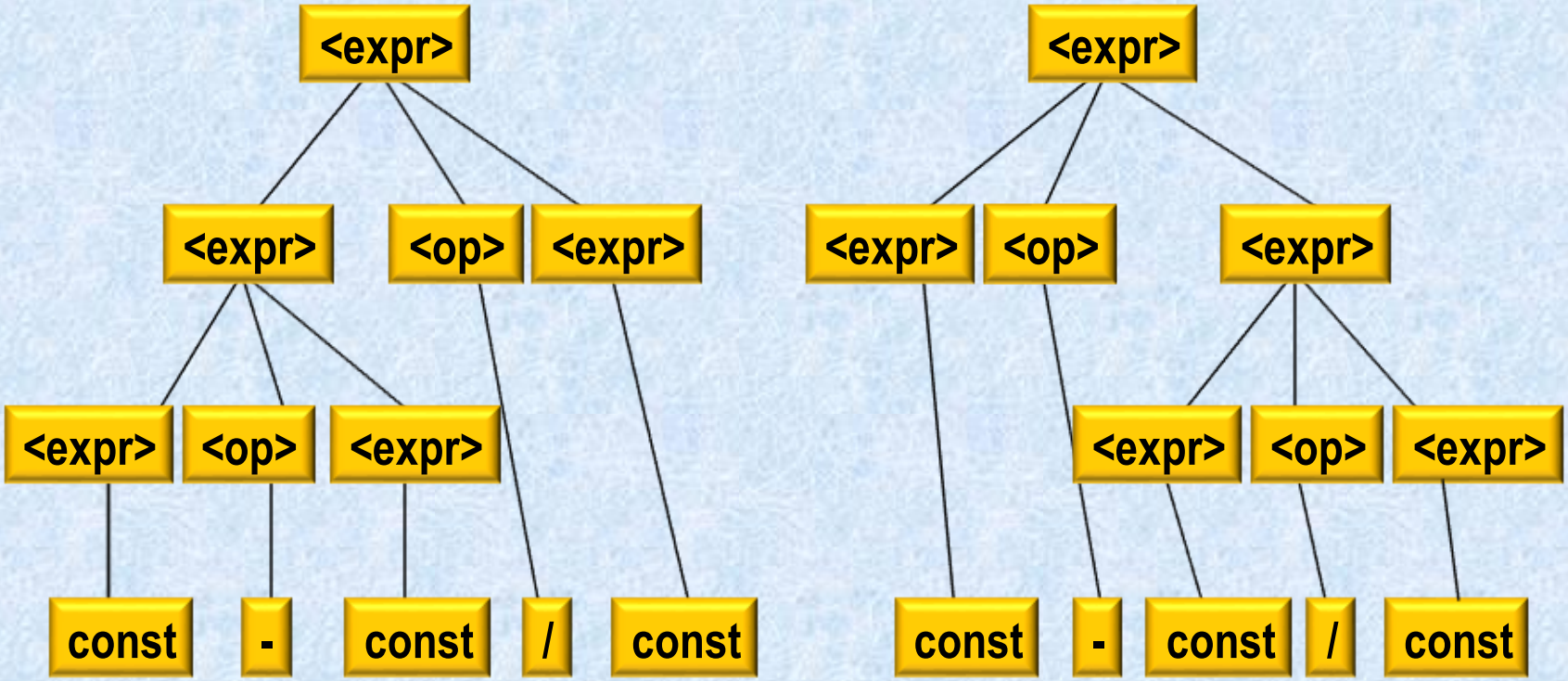
- Bir gramer ancak ve ancak iki veya daha fazla farklı ayrıştırma ağacı olan bir cümlesel biçim (sentential form) üretiyorsa *belirsiz*dir

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:

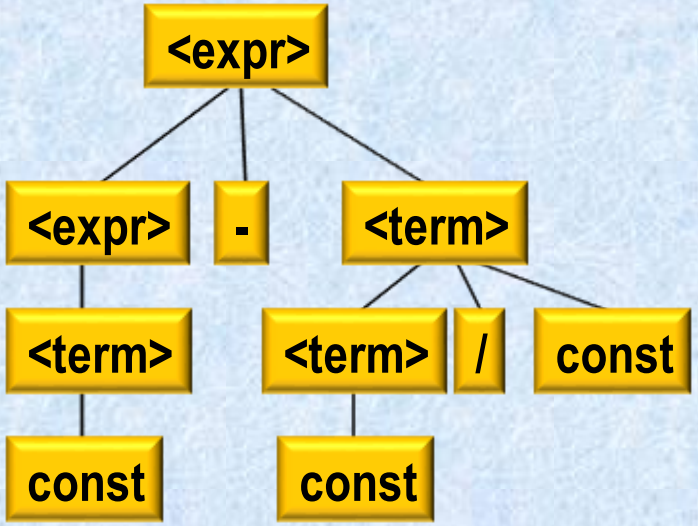
<expr> ➔ <expr> – <term>

➔ <term> – <term>

➔ const – <term>

➔ const – <term> / const

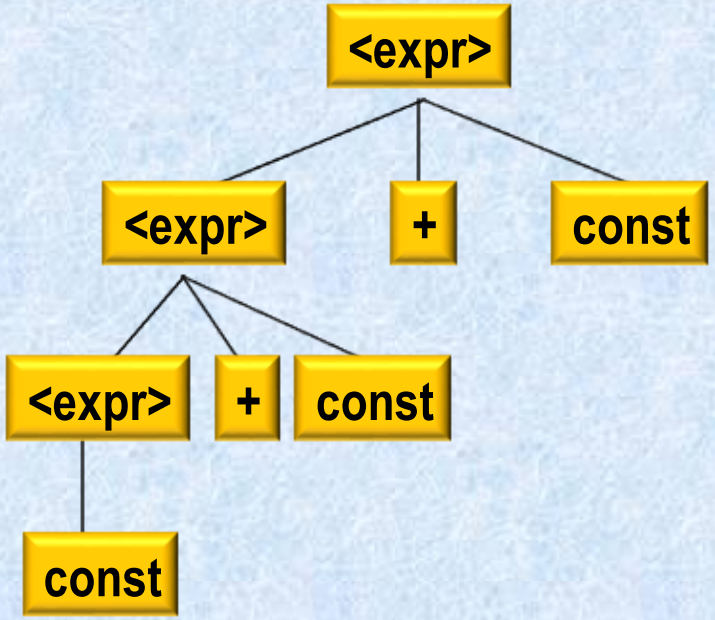
➔ const – const / const



Operatörlerin Birleşirliği (Associativity)

- Operatör birleşirliği (associativity) de gramerle gösterilebilir

```
<expr> -> <expr> + <expr> |  const  (Belirsiz)
<expr> -> <expr> + const  |  const  (Belirli, Kesin)
```

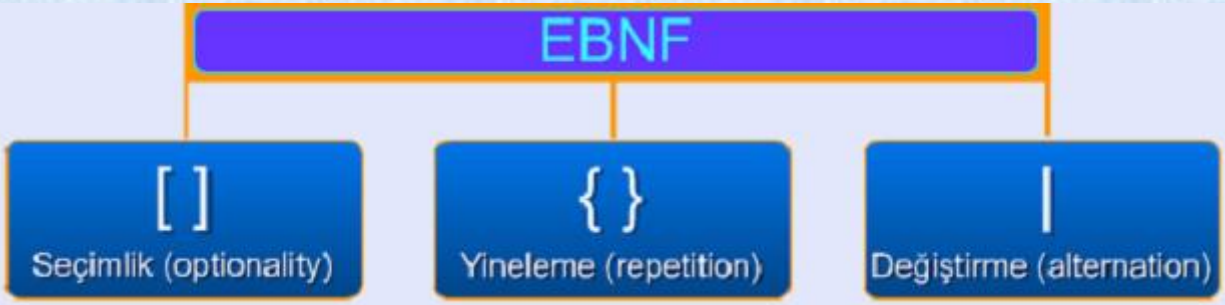


Genişletilmiş BNF (Extended BNF())

- 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:

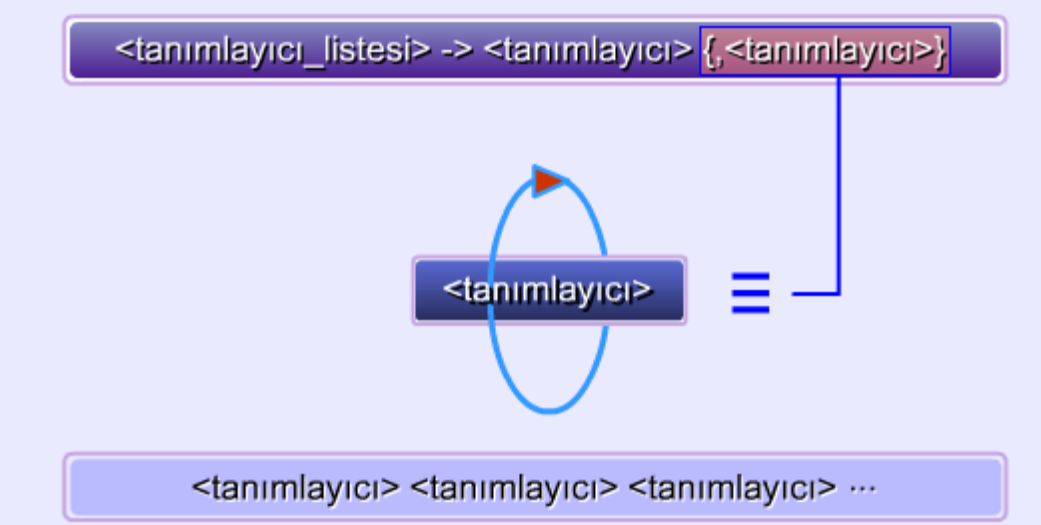
<seçimlik_deyim> -> If (<mantıksal>) <deyim> [else <deyim>];

- [ ] 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>
```

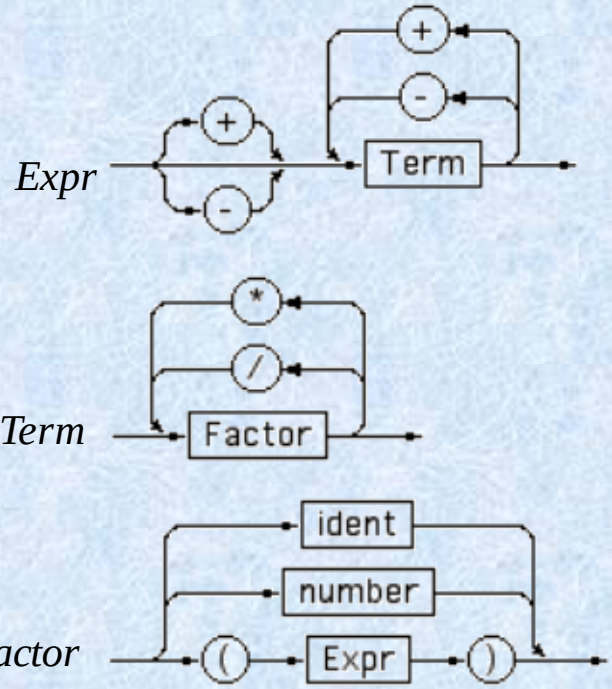
- **EBNF**

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

Arithmetik İfadelerin Grameri

Kurallar

```
Expr   = [ "+" | "-" ] Term { ( "+" | "-" ) Term }.
Term   = Factor { ( "*" | "/" ) Factor }.
Factor = ident | number | "(" Expr ")".
```



Terminal semboller

Basit Term. Sem.:	"+", "-", "*", "/", "(", ")" (sadece 1 örnek)
Terminal sınıfları:	ident, number (çoklu örnekler)

Terminal olmayan semboller

Expr, Term, Factor

Başlangıç sembolü

Expr

3.4 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

- Bir Ada prosedürünün **`end`**'inin üzerindeki ismin, prosedür ismiyle aynı olması kuralını, statik semantikle açıklamak için nitelik gramerlerini kullanalım
- Sentaks kuralı: <proc, def> → **`procedure`** <proc_name>[1]<proc_body> **`end`** <proc_name>[2]
- Semantik kuralı: <proc_name>[1].string= <proc_name>[2].string

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

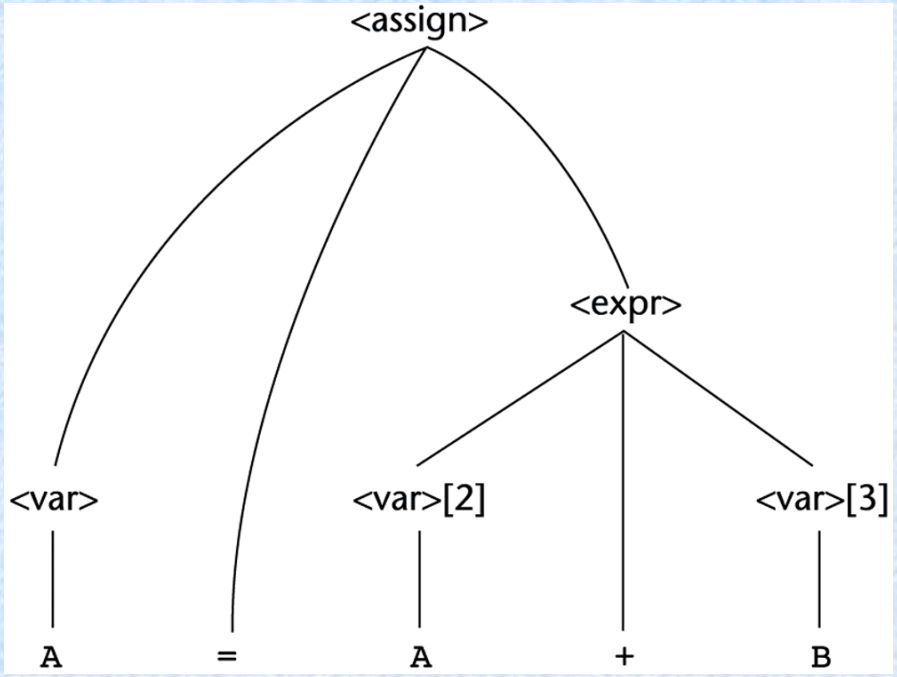
Cümle: A = A + B

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*

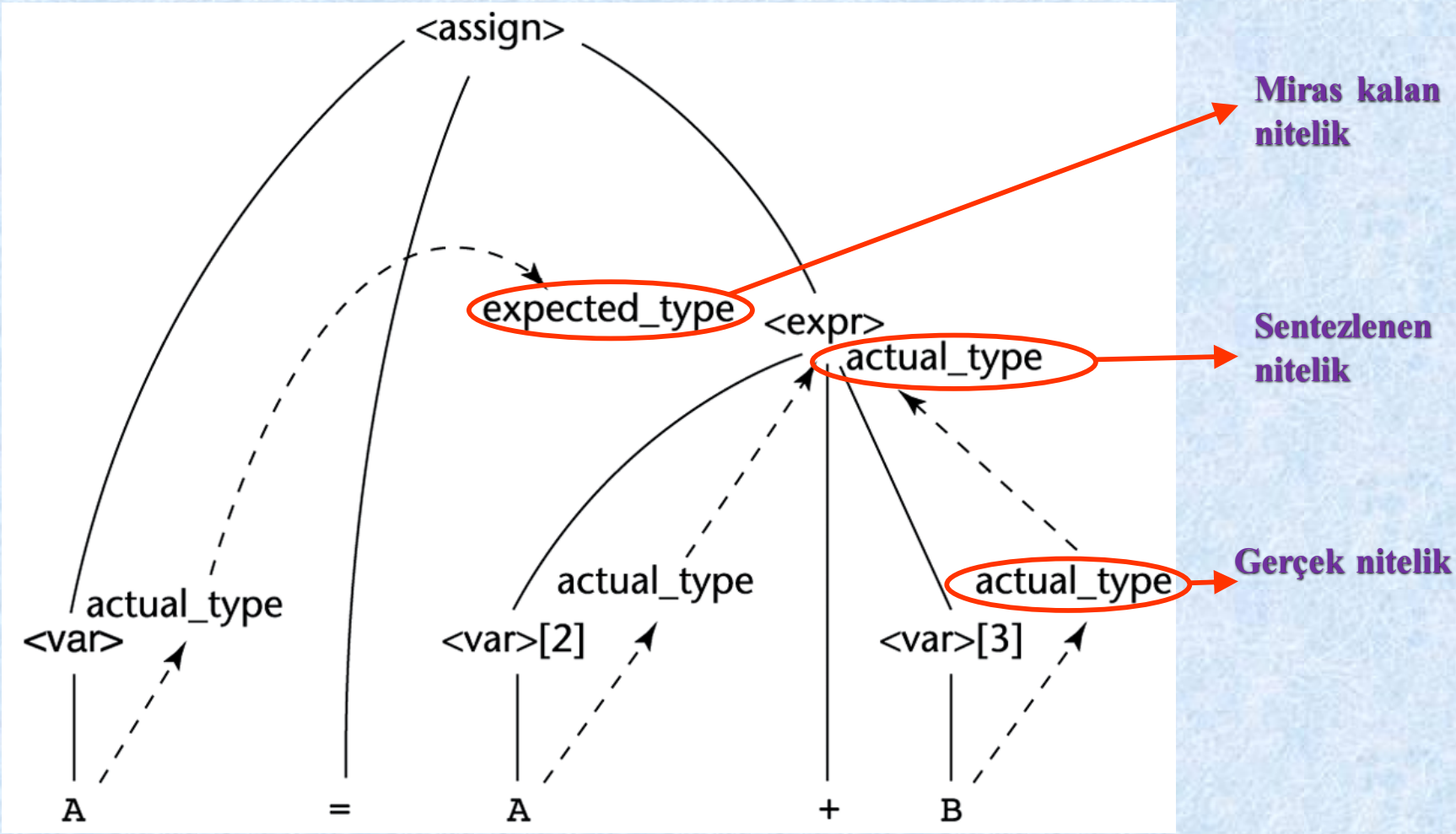
TRUE ya da FALSE'tur (Kural 2)

Gramer:

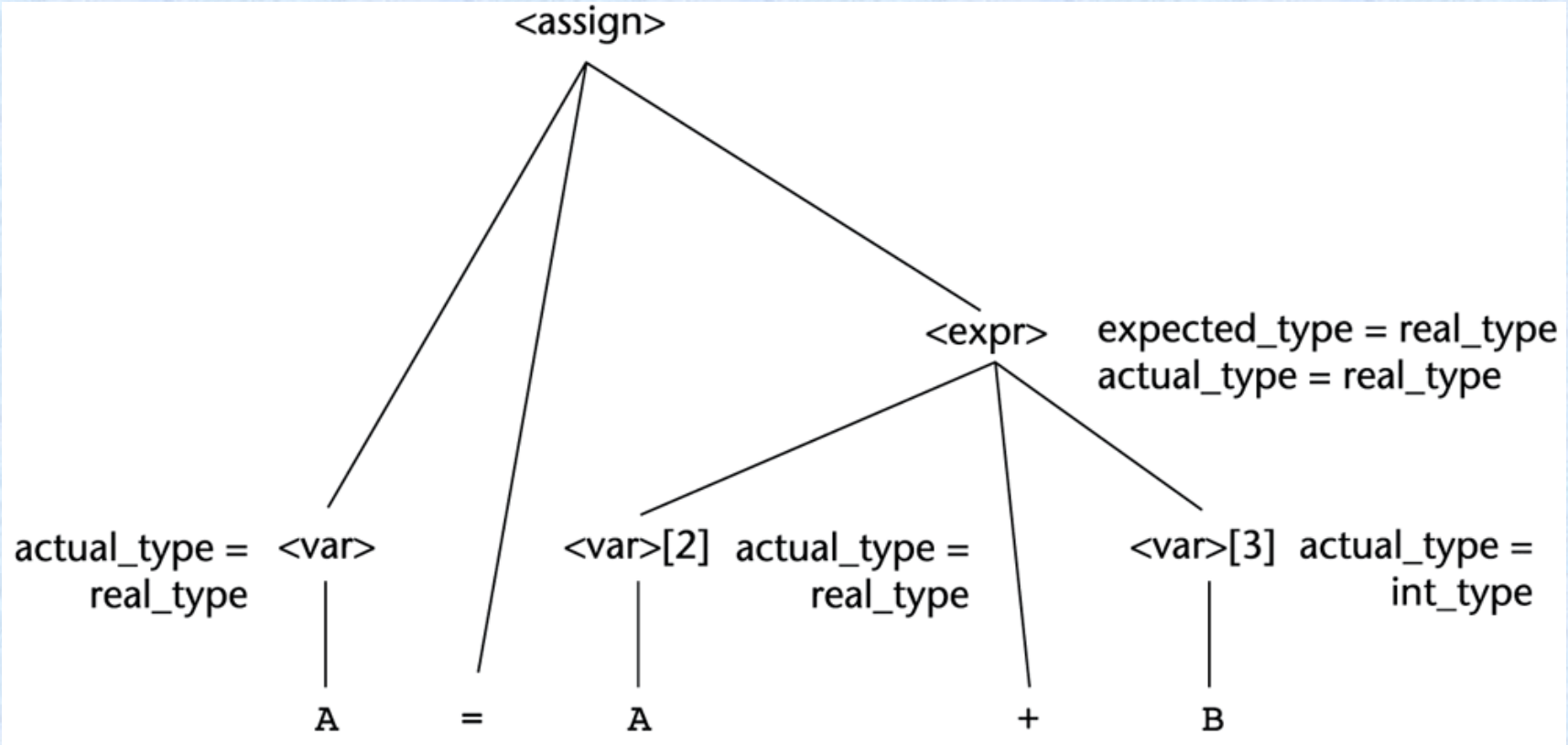
<assign> → <var> = <expr>
<expr> → <var> + <var> | <var>
<var> → A | B | C



Nitelik Değerlerini Hesaplama – Ayrıştırma Ağacında Nitelik Akışı



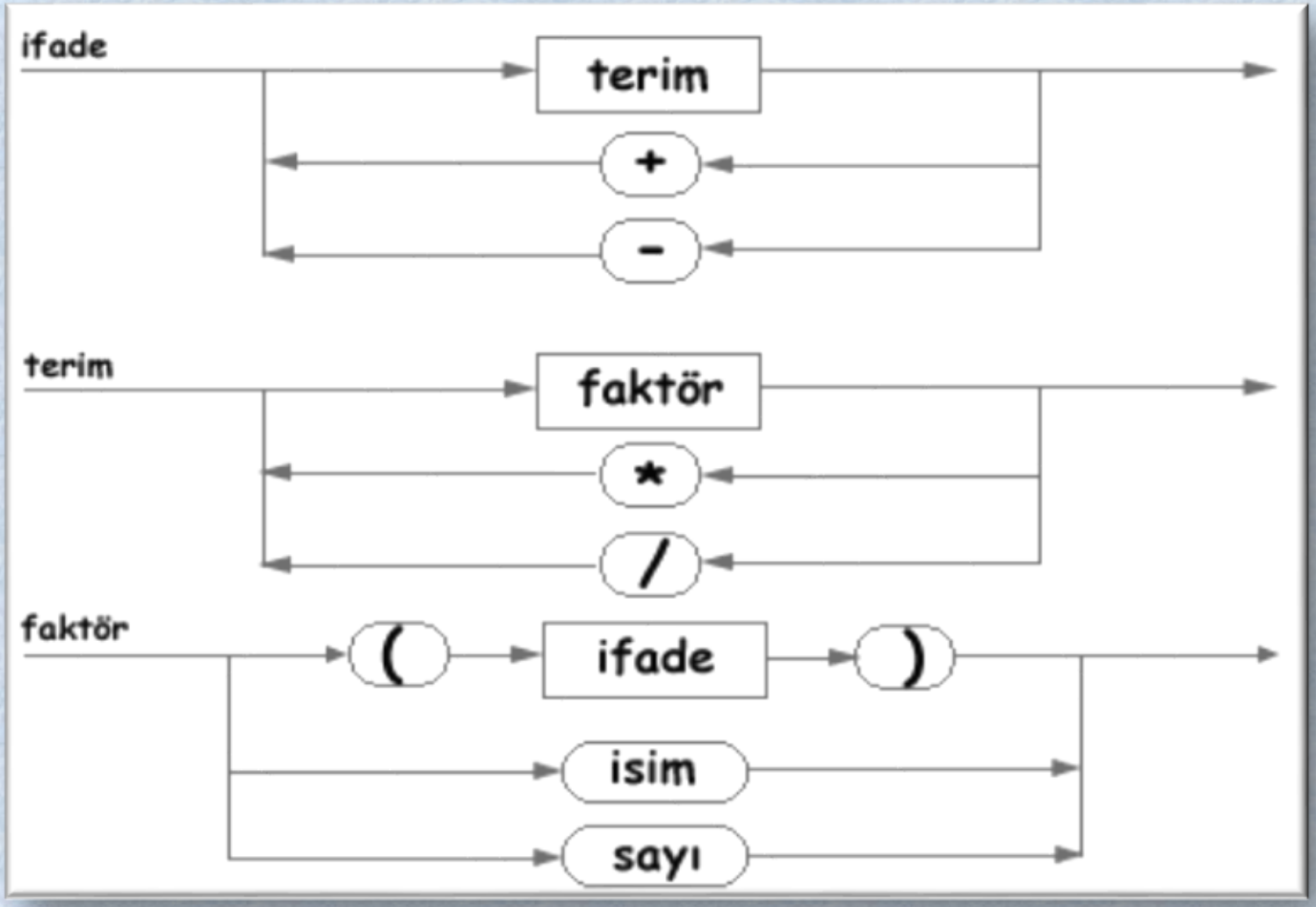
Nitelik Değerlerini Hesaplama – Tam bağlanmış nitelik ağacı



Sözdizim Grafikleri

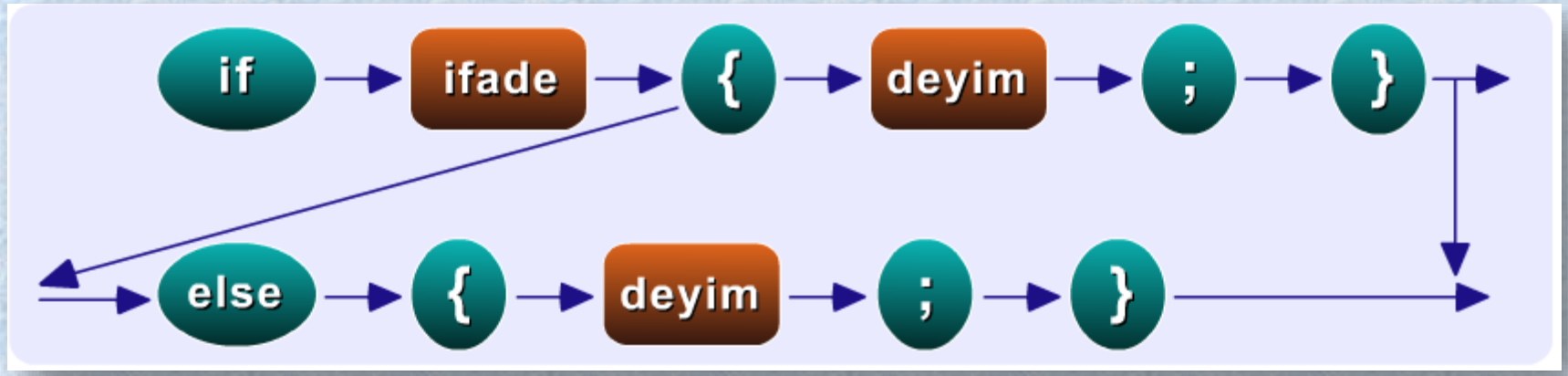
- BNF ve EBNF'teki kurallar, ayrıştırma ağacı dışında **sözdizim grafikleri** *(syntax graphs)* ile de gösterilebilir. Sözdizim grafikleri, ilk olarak Pascal'ın gramerini açıklamak için kullanılmıştır. BNF'ye eşdeğer olarak düşünülmüştür.
- Sözdizim grafiklerinde kurallar, düğümleri semboller olan iki-yönlü yönlendirilmiş grafikler ile gösterilir. Grafikteki olası yollar, kuraldaki terminal_olmayanı tanımlayan olası sembol sıralarını göstermektedir. Terminal semboller oval düğümler ile, terminal_olmayan semboller ise dikdörtgensel düğümlerle gösterilmektedir.

Sözdizim Grafikleri



Sözdizim Grafikleri

- Şekilde C'deki *if –then –else* deyiminin sözdizim grafiği görülmektedir.



3.5 Semantik

- Sözdizimsel olarak doğru olan tüm programların bir anlamı olmayabilir. Bir programın çalıştırılabilmesi için, hem söz dizim açısından hem de anlam açısından doğru olması gerekir.
- Bir programlama dilinin **anlam** *(semantik)* **kuralları**, bir dilde sözdizimsel olarak geçerli olan herhangi bir programın anlamını belirler.
- Dillerin sözdizimi BNF gibi ağaçlarla kolaylıkla tanımlanabilmesine rağmen, anlamlarının tanımlanması farklıdır.
- Semantiği tanımlamak için yaygın kabul edilmiş tek bir gösterim veya formalizm yoktur

Programlama Dillerinin Anlamsal (Semantik) Olarak Tanımlanması

Anlam tanımlama

- Anlamsal tanımlama için var olan yöntemler oldukça karmaşıktır ve hiçbir yöntem söz dizim tanımlamak için kullanılan BNF meta dili gibi yaygın kullanıma ulaşmamıştır.

- Anlam tanımlama için, dil yapılarının Türkçe gibi bir doğal dilde açıklanması sağlanır. Ancak doğal dil kullanılarak yapılan açıklamalar, açık ve kesin olmaz.

Programlama Dillerinin Anlamsal (Semantik) Olarak Tanımlanması

Durağan (Statik) Anlam Kuralları

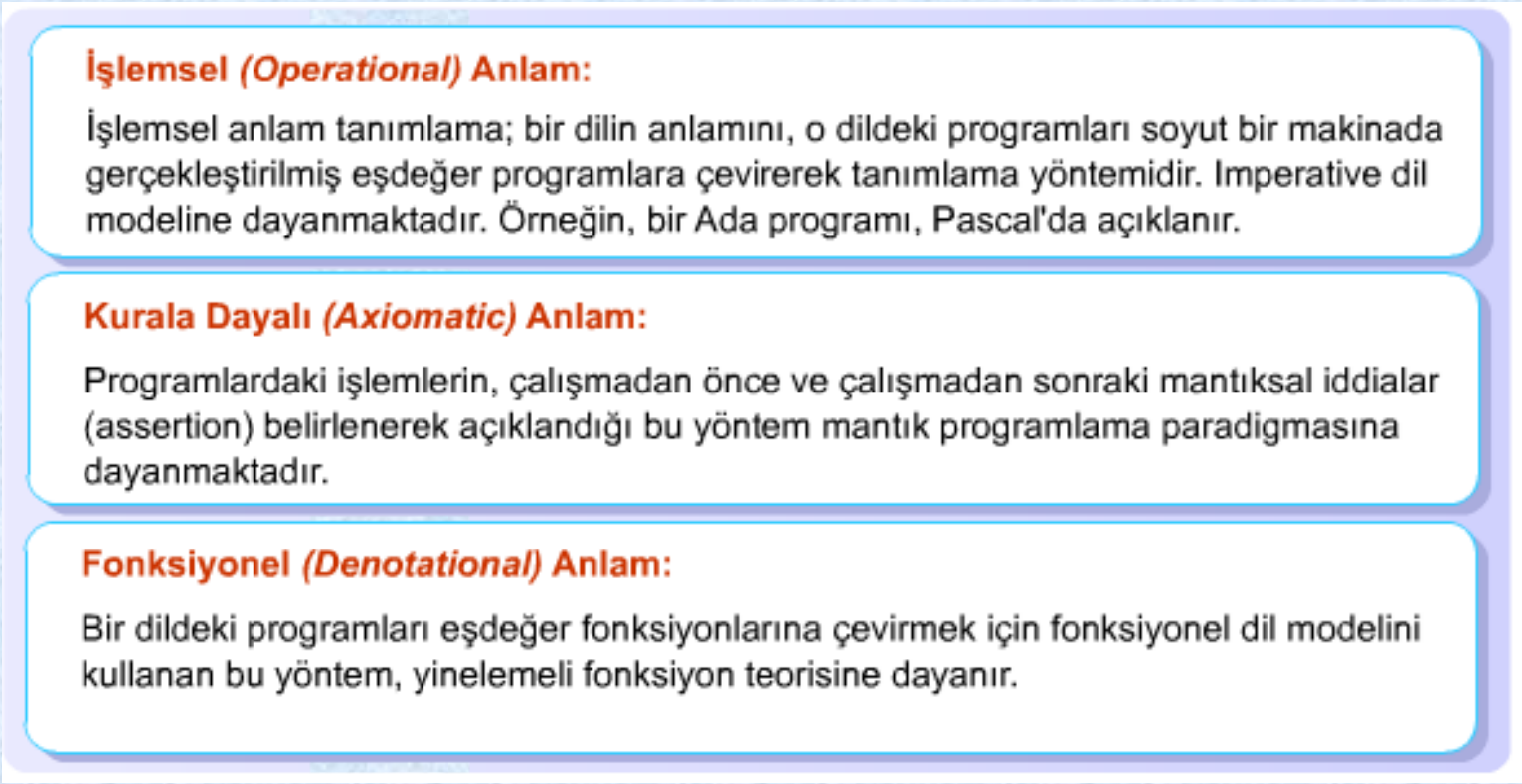
Programlama dillerinde bulunan bazı kuralların, BNF ile gösterilmesi olası değildir. Örneğin; tüm değişkenlerin kullanılmadan önce tanımlanmalarının gerekmesi gibi.
Bu ve benzeri kurallar, dilin durağan (static) anlam kurallarıdır.
Durağan anlam kuralları, programların derlenmesi sırasında kontrol edilebilir.

Dinamik Anlam Kuralları

Bir dildeki çeşitli yapıların çalıştırılmasının etkilerini açıklayan kurallara, dilin dinamik anlam kuralları denir.

Programlama Dillerinin Anlamsal (Semantik) Olarak Tanımlanması

- **Resmi Anlam Tanımlama**
- Resmi anlam tanımlama için kullanılacak metadil, tanımlamanın açık ve kesin olması için, iyi anlaşılmış sayısal kavramlara dayanmalıdır. Aşağıdaki çizimde resmi anlam tanımlama için kullanılan üç yaklaşım tanıtılmaktadır:



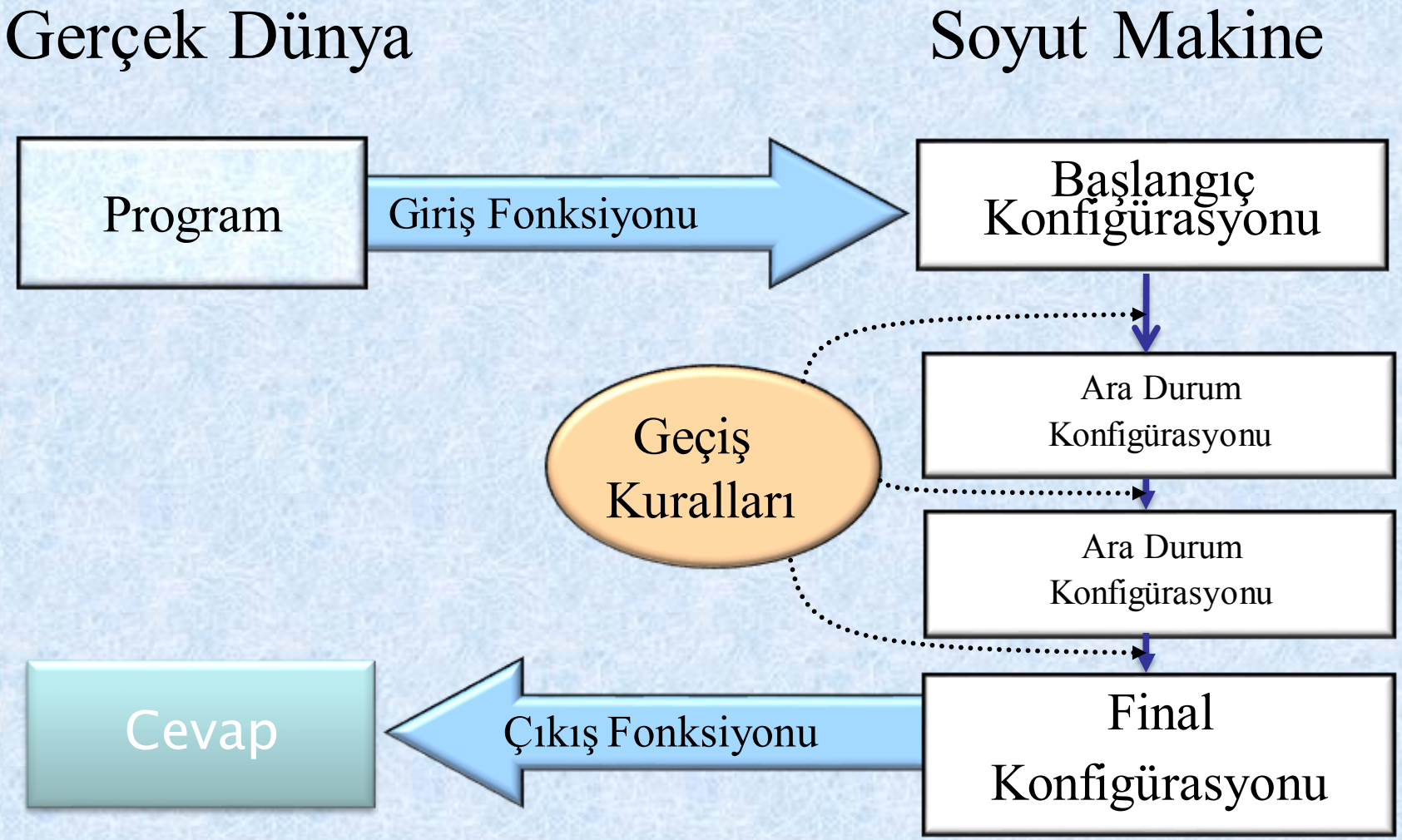
a) İşlemsel (Operational) Semantik

- Bir programı simulasyon veya gerçek olarak makine üzerinde çalıştırarak anlamını açıklamaktır. Makinenin durumundaki değişme (bellek, saklayıcılar (registers), vs.) ifadenin anlamını tanımlar
- Yüksek-düzeyli bir dil için işlemsel semantiği kullanmak için, bir sanal makine gereklidir
- Donanım saf yorumlayıcı çok pahalı olacaktır
- Yazılım saf yorumlayıcının bazı problemleri:
  - Bilgisayara özgü ayrıntılı özellikler faaliyetlerin anlaşılmasını zorlaştırır
  - Böyle bir semantik tanımı makine-bağımlı olurdu

İşlemsel (Operational) Semantik

- Daha iyi bir alternatif: Tam bir bilgisayar simulasyonu
- İşlem:
  - Bir çevirmen (translator) oluştur (kaynak kodunu, idealleştirilen bir bilgisayarın makine koduna çevirir)
  - İdealleştirilen bilgisayar için bir simülator oluştur
- İşlemsel semantiğin değerlendirmesi:
  - Informal olarak kullanılırsa iyidir (dil el kitapları, vs.)
  - Formal olarak kullanılırsa aşırı derecede karmaşıktır (örn., VDL), PL/I'ın semantiğini tanımlamak için kullanılıyordu.

İşlemsel (Operational) Semantik



Structured Operational Semantik

Bir dil için SOS beş-tuple'dır:

C	Soyut makine için konfigürasyon kümesi
$\Rightarrow$	Geçiş bağıntısı (C x C'in alt kümesi)
I	Program $\rightarrow C$ (giriş fonksiyonu)
$\mathcal{F}$	Final konfigürasyonlar kümesi
O	$\mathcal{F} \rightarrow$ Cevap (çıkış fonksiyonu)

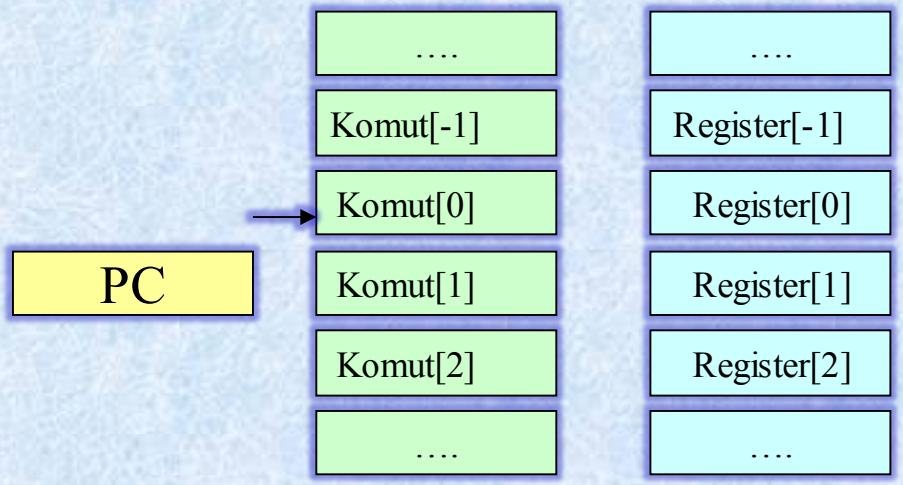
İşlemsel (Operational) Semantik

Soyut Makine:
Register Virtual Machine (RVM)

Konfigürasyon

- Komutlar dizisi
- Program sayacı (PC)
- Registerlarda değerler (herhangi integer)

ile tanımlanır



C = Komutlar x PC x RegisterFile

İşlemsel (Operational) Semantik

Giriş Fonksiyonu: I: Program $\rightarrow C$

C = Komutlar x PC x RegisterFile

0'dan n – 1'e kadar numaralı n komutlu Program için:

Komutlar[m] = Program[m] m >= 0 && m < için

Komutlar[m] = **ERROR** aksi halde

PC = 0

RegisterFile[n] = 0 tüm n tamsayılar için

İşlemsel (Operational) Semantik

Final Konfigürasyonları

$\mathcal{F}$ = Komutlar x PC x RegisterFile

Komutlar[PC] = **HALT**

Çıkış Fonksiyonu

$O: \mathcal{F} \rightarrow$ Cevap

Cevap= RegisterFile[0]'daki değer

İşlemsel (Operational) Semantik

Geçiş Kuralları Formu

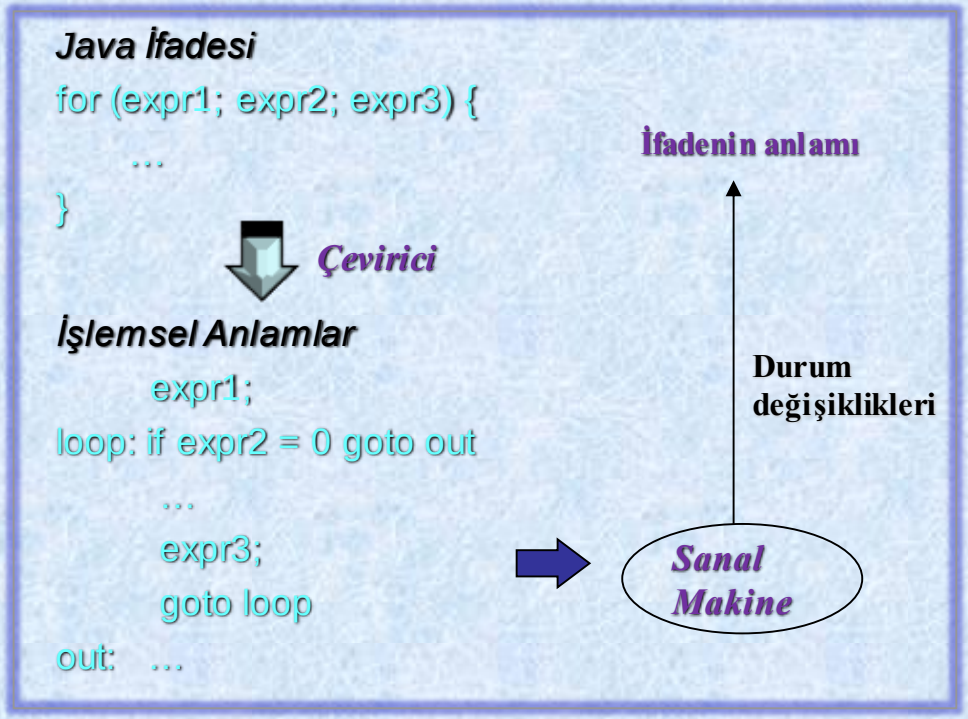
$$\frac{\textit{Antecedents}}{c \Rightarrow c'}$$

c, C'nin bir üyesi.

İşlemsel (Operational) Semantik

- İşlemsel anlamlarda, bir ifadenin ya da programın anlamını daha iyi anlamak için daha kolay anlaşılır bir dile çevirme işlemi vardır.
- İlk iş uygun bir ara dil geliştirmektir. Bu ara dilin temel karakteristiği basitliği ve açıklığıdır. Ara dildeki her yapı açık ve belirli olan anlama sahip olmalıdır. Örneğin;

C ifadesi	İşlemsel anlam
`for(expr1;expr2;expr3){`	**expr1;**
…	**loop: if expr2==0 goto out;**
`}`	…
	expr3;
	goto loop;
	out:….


Java İfadesi
for (expr1; expr2; expr3) {
...
}
Çevirici
İşlemsel Anlamlar
expr1;
loop: if expr2 = 0 goto out
...
expr3;
goto loop
out: ...
İfadenin anlamı
Durum değişiklikleri
Sanal Makine

b) Kurala Dayalı (Axiomatic) Semantik

- Biçimsel mantığa dayalıdır (predicate calculus)
- Orijinal amaç: Biçimsel program doğrulaması
- Yaklaşım: Dildeki her bir ifade tipi için aksiyomlar veya çıkarsama kuralları tanımlamak (deyimlerin (expressions) diğer deyimlere dönüştürülmesine imkan sağlamak için)
- Deyimlere iddia (assertions) adı verilir

Kurala Dayalı (Aksiyomatik) Semantik

- Bir ifadenin önündeki bir iddia (assertion) (bir önşart(precondition)), çalıştırıldığı zaman değişkenler arasında true olan ilişki ve kısıtları belirtir
- Bir ifadenin arkasından gelen iddiaya sonşart (postcondition) denir
- En zayıf önşart (weakest precondition), sonşartı garanti eden asgari kısıtlayıcı önşarttır

Kurala Dayalı (Aksiyomatik) Semantik

- Ön-son (Pre-post) biçimi :

  {P} ifade {Q}

- Bir örnek: $a = b + 1$ $\{a > 1\}$

  Mümkün bir önşart: $\{b > 10\}$

  En zayıf önşart: $\{b > 0\}$

Kurala Dayalı (Aksiyomatik) Semantik

Atama ifadeleri

- Önkoşul ve son koşul ifadelerin her ikisi de atama ifadesinin anlamını tam olarak tanımlamamızı sağlar.
- X=E genel bir atama ifadesi ve Q da onun son koşulu olsun. Önkoşul şöyle tanımlanır;
- $P=\{Q_{x->E}\}$ yani P, Q olan tüm X ler E ile değiştirilerek hesaplanır.

(x = E):

$\{Q_{x->E}\}$ x = E {Q}

Örnek:

x=2*y-3 {x>25} sonkoşul

2*y-3>25

y>14 olması gerekir. En zayıf ön şart {y>14} dir.

Kurala Dayalı (Aksiyomatik) Semantik

Atama ifadeleri

Örnek:

{x>5} x=x-3 {x>0}

x-3>0

x>3 en zayıf ön koşul x>3 burada {x>5} ön koşulunu da ima eder yani doğruluk kanıtlanıyor.

Kurala Dayalı (Aksiyomatik) Semantik

- Sonuç (Consequence) kuralı:

$$\frac{\{P\}\,S\,\{Q\},\ P' \Rightarrow P,\ Q \Rightarrow Q'}{\{P'\}\,S\,\{Q'\}}$$

$$\{x>3\}\ x = x - 3\ \{x>0\},\ (x>5)\Rightarrow(x>3),\ (x>0)\Rightarrow(x>0)$$

$$\{x>5\}\ x = x - 3\ \{x>0\}$$

Kurala Dayalı (Aksiyomatik) Semantik

Seçim (Selection)

- Seçim ifadeleri için sonuç çıkarma kuralı
- if B then S1 else S2

$$\frac{\{B \text{ and } P\}\, S1\, \{Q\},\ \{(\text{not } B) \text{ and } P\}\, S2\, \{Q\}}{\{P\} \text{if } B \text{ then } S1 \text{ else } S2\, \{Q\}}$$

- Bu kural seçim ifadesinin mantıksal kontrol ifadesinin doğru ya da yanlış olduğu durumlarda da kanıtlanması gerektiğini söyler. Bunlar için kullanacağımız bir ortak {P} koşuluna ihtiyaç vardır. Örneğin

  if (x>0) y=y-1
  else y=y+1

  bu ifade için son koşul {Q}={y>0} olsun

Kurala Dayalı (Aksiyomatik) Semantik

Seçim (Selection)

y=y-1 {y>0}

y-1>0 {y>1} bu then kısmı için {P} olarak kullanılabilir. Şimdi else yi uygulayalım

y=y+1

y+1>0 {y>-1} {y>1}=>{y-1} olduğundan {y>1} her ikisi içinde ön koşuldur.

Kurala Dayalı (Aksiyomatik) Semantik

Mantıksal testi önde olan koşullar

while döngüleri buna örnektir. Döngü sayısı belirli olmadığından ön koşulun hesaplanması kolay değildir. Döngü sayısı belirliyse arka arkaya eklenmiş ifadeler dizisi gibi düşünülüp en son elamanın sonkoşulu kullanılarak yukarıya doğru gidilir ve en sonunda önkoşul bulunur.

Kurala Dayalı (Aksiyomatik) Semantik

Örnek:

while y<>x do y=y+1 end {y=x}

Bu döngüde hiç döngü olmasa en zayıf ön koşul {y=x}

İlk iterasyon

(y=y+1, {y=x}) = {y+1=x}, ya da {y=x-1}

İki iterasyon için

(y=y+1, {y=x}) = {y+1=x-1}, ya da {y=x-2}

üç iterasyon için

(y=y+1, {y=x}) = {y+1=x-2}, ya da {y=x-3}

Burada {y<x} olduğu açıktır. Bunu hiç iterasyon olmadığı durumu da birleştirirsek ön koşul {y<=x} olur.

Kurala Dayalı (Aksiyomatik) Semantik

- Bir programlama dilinin tamamının aksiyomatik metot kullanılarak anlamını tanımlamak oldukça zordur. Her ifade için aksiyom ve sonuç çıkarım kurallarının tanımlanması gerekir, bu her zaman mümkün değildir.

Kurala Dayalı (Aksiyomatik) Semantik

- Sıralar (sequences) için çıkarsama kuralı

  Bir S1; S2 sırası (bitişik program sırası) için:

  {P1} S1 {P2}

  {P2} S2 {P3}

Çıkarsama kuralı: $$\frac{\{P1\}\,S1\,\{P2\},\{P2\}\,S2\,\{P3\}}{\{P1\}\,S1;S2\,\{P3\}}$$

Örnek:

y = 3 * x + 1; x = y + 3; { x < 10 }

2. ifadenin önşartı: y < 7

1. ifadenin önşartı: x < 2

Kurala Dayalı (Aksiyomatik) Semantik

- Mantıksal öntest döngüleri için bir çıkarsama kuralı

  Döngü yapısı için:

  {P} while B do S end {Q}

  çıkarsama kuralı:

$$\frac{(\text{I and B})\ \text{S}\ \{\text{I}\}}{\{\text{I}\}\ \text{while B do S}\ \{\text{I and (not B)}\}}$$

  I döngü sabiti (invariant) ise (tümevarımsal hipotez (inductive hypothesis))

Kurala Dayalı (Aksiyomatik) Semantik

- Döngü sabitinin (invariant) özellikleri

  I, aşağıdaki şartları sağlamalıdır:

  - P => I (döngü değişkeni başlangıçta true olmalı)
  - {I} B {I} (Boolean hesabı I'nin doğruluğunu(geçerliliğini) değiştirmemelidir)
  - {I and B} S {I} (Döngü gövdesinin çalıştırılmasıyla I değişmez)
  - (I and (not B)) => Q (I true ise ve B false ise, Q bulunur (implied))
  - Döngü sonlanır (ispatlamak zor olabilir)

Kurala Dayalı (Aksiyomatik) Semantik

- Döngü sabiti I, döngü sonşartının zayıflatılmış bir sürümüdür, ve aynı zamanda bir önşarttır.
- I, döngünün başlamasından önce yerine getirilebilecek kadar zayıf olmalıdır, fakat döngü çıkış şartıyla birleştirildiğinde, sonşartın doğru olmasını zorlayacak kadar güçlü olmalıdır

Kurala Dayalı (Aksiyomatik) Semantik

- Program ispat işlemi: Bütün program için sonşart istenen sonuçtur. Programda ilk ifadeye kadar geriye doğru çalışılır. Birinci ifadedeki önşart program şartnamesiyle (spec.) aynıysa, program doğrudur.

Kurala Dayalı (Aksiyomatik) Semantik

Program ispatları

{x=a and y=b}

t=x

x=y

y=t

{x=b and y=a}

y=t {x=b and y=a} dan {x=b and t=a} ön koşulu bulunur

x=y {x=b and t=a} dan {y=b and t=a} ön koşulu bulunur

t=x {y=b and t=a} dan {y=b and x=a} ön koşulu bulunur

Bu da yazılan önşart ile aynı olduğunu ispatlamış olur.

Kurala Dayalı (Aksiyomatik) Semantik

- Aksiyomatik Semantiğin değerlendirilmesi:
  - Bir dildeki bütün ifadeler için aksiyom ve çıkarsama kuralları geliştirmek zordur
  - İspatların doğruluğu için iyi bir araçtır, ve programlama mantığı için mükemmel bir çatıdır, fakat dil kullanıcıları ve derleyici yazanlar için kullanışlı değildir
  - Programlama dilinin anlamını tanımlamadaki yararlılığı dil kullanıcıları veya derleyici yazarları için sınırlandırılmıştır

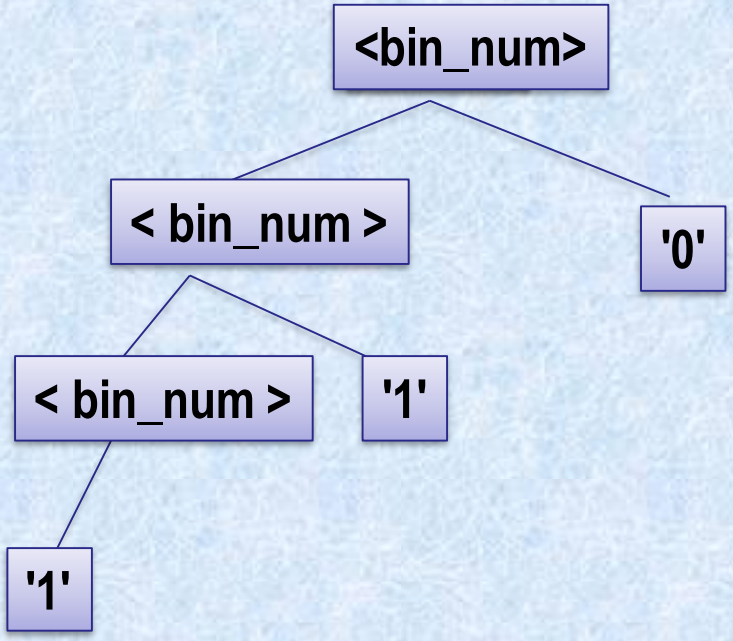
c) Fonksiyonel (Denotasyonel) Semantik

- Scott ve Strachey (1970) tarafından geliştirilmiştir
- Özyinelemeli (recursive) fonksiyon teorisine dayalıdır. Temel dildeki her varlık için hem bir matematiksel nesne hem de fonksiyonel tanımlama vardır.
- En soyut ve programların anlamını tanımlamada kullanılan en yaygın bir semantik tanımlama metodudur
- Bu fonksiyon bir varlığın örneklerini, matematiksel modelin örnekleri ile eşleştirir. Nesneler tanımlandığından, karşılık geldikleri varlıkların anlamlarını tanımlar.
- İkili (binary) sayı gösteren gramer şöyle tanımlanabilir.

<bin_num> → '0' | '1' | <bin_num> '0' | <bin_num> '1'

Fonksiyonel (Denotasyonel) Semantik

110 içi ayrıştırma ağacı



Fonksiyonel (Denotasyonel) Semantik

- Dil için denotasyonel şartnameler (spec) oluşturma işlemidir (kolay sayılmaz):
  - Her dil varlığı için matematiksel bir nesne tanımlanır
  - Dil varlıklarının örneklerini karşılık gelen matematiksel nesnelerin örneklerine eşleştiren bir fonksiyon tanımlanır
- Dil yapılarının anlamları sadece programın değişkenlerinin değerleriyle tanımlanır

Fonksiyonel (Denotasyonel) Semantik

- Denotasyonel ve işlemsel (operational) semantik arasındaki fark: İşlemsel semantikte, durum değişimleri kodlanmış algoritmalarla tanımlanır ; denotasyonel semantikte, sıkı matematiksel fonksiyonlarla tanımlanır

Fonksiyonel (Denotasyonel) Semantik

- Programın durumu bütün güncel değişkenlerinin değerleridir

  $s = \{<i_1, v_1>, <i_2, v_2>, \ldots, <i_n, v_n>\}$

- **VARMAP** öyle bir fonksiyon olsun ki, bir değişkenin adı ve durumu verildiğinde, o değişkenin güncel değerini döndürsün

  $\mathbf{VARMAP}(i_j, s) = v_j$

Fonksiyonel (Denotasyonel) Semantik

- İkili sayıların anlamlarını tanımlamak için fonksiyonel anlamları ve gramer kurallarını kullanırsak; gerçek anlamı sağ tarafında tek bir terminal sembol olan kural için tanımlarız. Bu örnekte anlamlı nesneler ilk iki kural için ilişkilendirilmelidir. Nesnelerin anlamsal değerleri N negatif olmayan ondalık tamsayılar olsun . Bu nesneler ile ikili sayıları eşleştireceğiz. M_{bin} anlamsal fonksiyon söz dizim nesneleri, N nesneleri ile eşler.

M_{bin}('0') = 0,

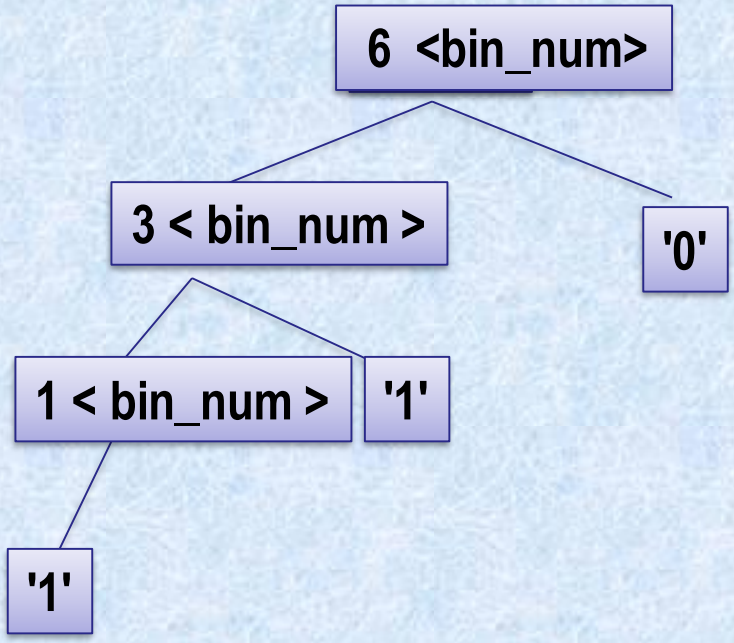
M_{bin}('1') = 1,

M_{bin} (< bin_num> '0') = 2 * M_{bin} (< bin_num>)

M_{bin} (< bin_num> '1') = 2 * M_{bin} (<bin_num>) + 1

Fonksiyonel (Denotasyonel) Semantik

- Fonksiyonel nesneler, yani anlamlar ayrıştırma ağacına şöyle atanır.



Fonksiyonel (Denotasyonel) Semantik

- Ondalık sayılar
  - Şu denotasyonel semantik tanımı, string sembollerden oluşan ondalık sayıları sayısal değerlere eşleştirir

<dec_num> → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
| <dec_num> (0 | 1 | 2 | 3 | 4 |
5 | 6 | 7 | 8 | 9)

M_{dec}('0') = 0, M_{dec} ('1') = 1, …, M_{dec} ('9') = 9
M_{dec} (<dec_num> '0') = 10 * M_{dec} (<dec_num>)
M_{dec} (<dec_num> '1') = 10 * M_{dec} (<dec_num>) + 1
…
M_{dec} (<dec_num> '9') = 10 * M_{dec} (<dec_num>) + 9

Fonksiyonel (Denotasyonel) Semantik

- Deyimler (Expressions)
  - Deyimleri $Z \cup \{error\}$ üzerine eşleştirir
  - Deyimleri, ondalık sayılar, değişkenler, veya bir aritmetik operatör ve her biri bir deyim olabilen iki operanda sahip ikili deyimler olarak varsayıyoruz.

Fonksiyonel (Denotasyonel) Semantik

```
Me(<expr>, s) Δ=
    case <expr> of
      <dec_num> => Mdec(<dec_num>, s)
      <var> =>
           if VARMAP(<var>, s) == undef
                then error
                else VARMAP(<var>, s)
     <binary_expr> =>
          if (Me(<binary_expr>.<left_expr>, s) == undef
               OR Me(<binary_expr>.<right_expr>, s) =
                                 undef)
             then error
      else
          if (<binary_expr>.<operator> == '+' then
             Me(<binary_expr>.<left_expr>, s) +
                    Me(<binary_expr>.<right_expr>, s)
           else Me(<binary_expr>.<left_expr>, s) *
              Me(<binary_expr>.<right_expr>, s)
      …
```

Fonksiyonel (Denotasyonel) Semantik

- Atama ifadeleri
  - Durum kümelerini durum kümelerine eşleştirir

$$
\begin{aligned}
&\mathbf{Ma(x := E,\ s)}\ \Delta= \\
&\quad \mathbf{if\ Me(E,\ s) == error} \\
&\qquad \mathbf{then\ error} \\
&\qquad \mathbf{else\ s'} = \\
&\{\langle i_1', v_1'\rangle, \langle i_2', v_2'\rangle, \ldots, \langle i_n', v_n'\rangle\}, \\
&\qquad\qquad \mathbf{where\ for\ j = 1,\ 2,\ \ldots,\ n,} \\
&\qquad\qquad\quad v_j' = \mathbf{VARMAP}(i_j,\ \mathbf{s})\ \mathbf{if}\ i_j \neq x \\
&\qquad\qquad\qquad\quad = \mathbf{Me(E,\ s)\ if}\ i_j == x
\end{aligned}
$$

Fonksiyonel (Denotasyonel) Semantik

- Mantıksal Öntest Döngüleri (Logical Pretest Loops)
  - Durum kümelerini durum kümelerine eşleştirir

M_l(while B do L, s) Δ=
    if M_b(B, s) == undef
        then error
        else if M_b(B, s) == false
            then s
            else if M_{sl}(L, s) == error
                then error
                else M_l(while B do L, M_{sl}(L, s))

Fonksiyonel (Denotasyonel) Semantik

- Döngünün anlamı; program değişkenlerinin, döngüdeki ifadelerin belirtilen sayıda ve hata olmadığını varsayarak çalıştırılmasından sonra aldığı değerleridir
- Esasında döngü, iterasyondan özyinelemeye dönüştürülmüştür, rekürsif kontrol matematiksel olarak diğer rekürsif durum eşleştirme fonksiyonlarıyla tanımlanır
- Özyineleme, iterasyonla karşılaştırıldığında, matematiksel kesinliklerle güçlüklerle (rigor) açıklaması daha kolaydır

Fonksiyonel (Denotasyonel) Semantik

- Denotasyonel semantiğin değerlendirilmesi
  - Programların doğruluğunu ispatlama için kullanılabilir
  - Programlar hakkında düşünmek için sıkı (kesin) (rigorous) bir yol sağlar
  - Dil tasarımında yardımcı olabilir
  - Derleyici üretme sistemlerinde kullanılmıştır
  - Karmaşıklığı yüzünden, dil kullanıcıları tarafından çok az kullanılmıştır

Semantik Belirleme Yöntemleri (Özet)

Operational Semantik:

- [[program]] = *soyut makine programı*
- Gerçekleştirme kolay
- Muhakeme zor

Denotational Semantik:

- [[program]] = *matematiksel ifade*
- (tipik olarak, bir fonksiyon)
- Muhakemeyi kolaylaştırır
- Uygun semantik alanları bulmak her zaman kolay değil

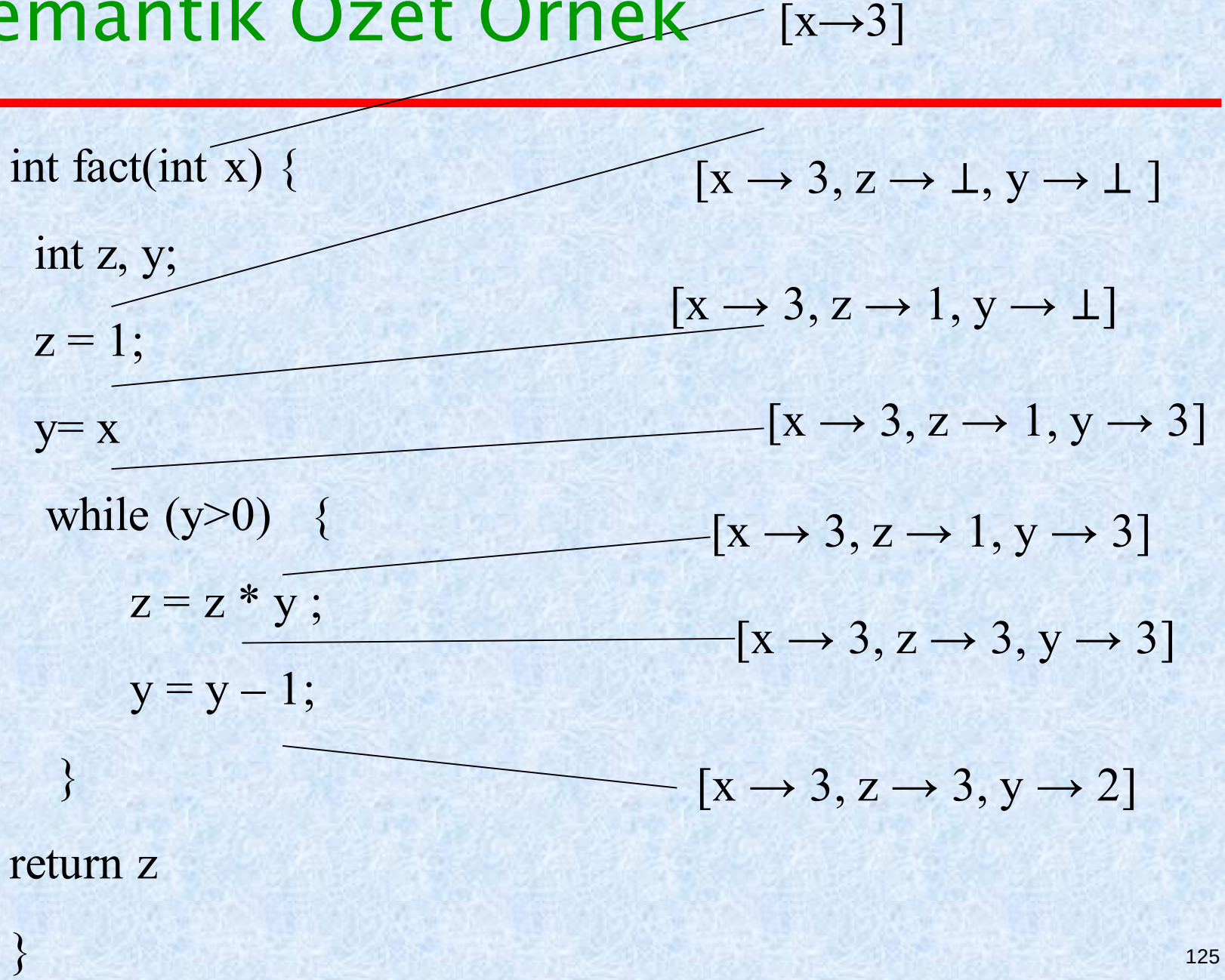
Axiomatic Semantik:

- [[program]] = *özellikler kümesi*
- Programlar hakkında teorem ispatı için iyi
- Uygulamadan biraz uzak

Structured Operational Semantik:

- [[program]] = *geçiş sistemi*
- (çıkarsama kuralları kullanarak tanımlanmıştır)
- Uyumluluk ve non non-determinizm için iyi
- Eşitlik hakkında muhakeme etme zor

Semantik Özet Örnek



Semantik Özet Örnek

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
         z = z * y ;
         y = y – 1;
    }
return z
}
```

$[x \rightarrow 3, z \rightarrow 3, y \rightarrow 2]$

$[x \rightarrow 3, z \rightarrow 3, y \rightarrow 2]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 2]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 1]$

Semantik Özet Örnek

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x
   while (y>0)   {
         z = z * y ;
         y = y – 1;
    }
return z
}
```

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 1]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 1]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 1]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 0]$

Semantik Özet Örnek

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x                                   [x → 3, z → 6, y → 0]
  while (y>0)   {
        z = z * y ;
        y = y – 1;
   }
return z ——— [x → 3, z → 6, y → 0]
}
```

Semantik Özet Örnek

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x;
   while (y>0)   {
         z = z * y ;
         y = y – 1;
    }
return 6
}
```

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 0]$

$[x \rightarrow 3, z \rightarrow 6, y \rightarrow 0]$

Örnek: Denotational Semantik

```
int fact(int x) {
  int z, y;
  z = 1;
  y= x ;
   while (y>0)    {
          z = z * y ;
          y = y - 1;
    }
return z;
}
```

$f=\lambda x.$ if x = 0 then 1 else x* f(x -1)

Örnek: Axiomatic Semantik

```
{x=n}
 int fact(int x) {  int z, y;
 z = 1;
{x=n ∧ z=1}
 y= x
{x=n ∧ z=1 ∧ y=n}
  while
    {x=n  ∧ y >0 ∧ z=n! / y!}
 (y>0)  {
       {x=n ∧ y >0 ∧  z=n! / y!}
        z = z * y ;
      {x=n ∧ y>0 ∧  z=n!/(y-1)!}
        y = y – 1;
      {x=n ∧ y >0 ∧  z=n!/y!}
   } return z} {x=n ∧ z=n!}
```

Örnek: Operational Semantik

yüksek-seviye program

```
for (e1; e2; e3) {
     .
     .
     .
}
<rest of program>
```

operational semantik

```
     e1;
L1:  if not(e2) goto L2
     .
     .
     .
     e3;
     goto L1
L2:  <rest of program>
```

Özet

- BNF ve içerik-bağımsız gramerler eşdeğer meta-dillerdir
  - Programlama dillerinin sentaksını tanımlayabilmek için uygundur
- Özellik grameri bir dilin hem sentaksını hem de semantiğini tanımlayabilen tanımlayıcı bir formalizmdir
- Semantik tanımının birincil metotları
  - İşlem (Operation), aksiyomatik, denotasyonel

Son Not: Programlama Dillerinin Standartlaştırılması

- Bir dil yaygın olarak kullanılmaya başlandıktan sonra, çeşitli donanım ortamlarına uyumlu dil gerçekleştirimlerinin üretilebilmesi için dilin sözdizimi ve anlamı tam ve açık olarak tanımlanmalı ve **dil standardı** oluşturulmalıdır.
- Bir dil standardı, dil tasarımcısı, dil tasarımını destekleyen kuruluş veya Amerika Birleşik Devletleri'nde *American National Standards Institute (**ANSI**)* gibi bir kuruluş tarafından tanımlanabilir. Benzer şekilde *International Standards Organization (**ISO**)* tarafından gerçekleştirilebilir

ÖDEVLER

1. BNF ve EBNF hakkında detaylı bilgi toplayınız
2. Java Diline ait BNF yapısını araştırınız.
3. a=a*(b+(c*a)) örneği için grameri ve sola dayalı türetme özelliğini gösterip ayrıştırma ağacını çiziniz.
4. Semantik analiz için kullanılan yöntemleri (İşlemsel, Yapısal İşlemsel, Kurala Dayalı ve Fonksiyonel) araştırıp örnekler ile destekleyiniz.

Hazırlayacağınız belgeler – doc ya da ppt formatında olacaktır

Kaynaklar

- Roberto Sebesta, Concepts Of Programming Languages, International 10th Edition 2013
- David Watt, Programming Language Design Concepts, 2004
- Michael Scott, Programming Languages Pragmatics, Third Edition, 2009
- Zeynep Orhan, Programlama Dilleri Ders Notları
- Mustafa Şahin, Programlama Dilleri Ders Notları
- Ahmet Yesevi Üniversitesi, Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları
- David Evans, Programming Languages Lecture Notes
- O. Nierstrasz, Programming Languages Lecture Notes
- Manuel E. Bermúdez, Programming Languages Lecture Notes
- Pranava K. Jha, Programming Languages Concepts Lecture Notes