-PROGRAMLAMA DİLLERİ DERSİ- (2018 DERS NOTLARI)

* Programlamaya Dilleri (Bölüm 0)
* Programlama Dillerine Giriş (Bölüm 1)
* Programlama Dillerinin Gelişimi (Bölüm 2)
* Sentaks ve Semantiği Tanımlama (Bölüm 3)
* Sözcüksel ve Sentaks Analiz (Bölüm 4)
* Adlar, Bağlama, Tip Kontrolü ve Kapsam (Bölüm 5)
* Veri Tipleri (Bölüm 6)
* Alt Programlar (Bölüm 8)
* Çıkmış Sorular

-HELİN YARDIMCI-

BÖLÜM - 0

* Bir programlama dili, bir problemin çözümünün bilgisayardaki gerçekleştirimini ifade etmek amacıyla programlar oluşturulması için kullanılan bir dildir.

Programlama dilinin dilbilgisi kuralları ⟹ Sözdizim (syntax)

Programlama Dili → Derleme → Makine Dili

Makine Dili ⇒ Bir bilgisayarın doğrudan anladığı gösterim olup, bilgisayarların ana dili olarak nitelendirilir. (Derleme ve yorumlama ile olur)

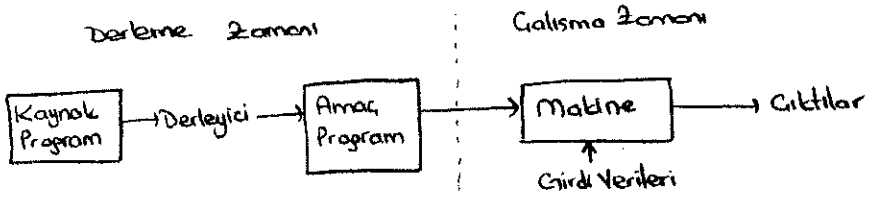
- Dil Çevrimi için derleme ve yorumlama olarak adlandırılan iki temel yöntem vardır.

Yorumlayıcı, bir programın her deyimini birer birer makine diline çevirir ve o deyimle ilgili bir altprogram çağırarak deyimin çalıştırılmasını sağlar.

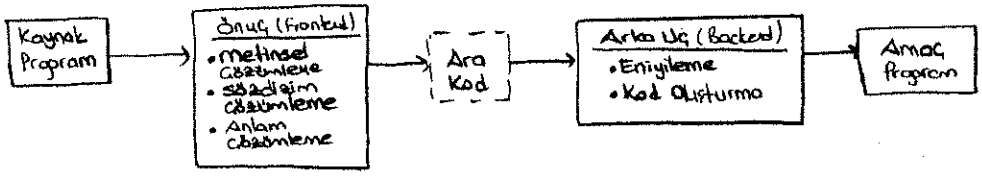
Yorumlayıcının genel çalışma şekli; Tüm programı deyim deyim denetler.
Bir döngü içindeki tüm deyimler her kerasında çevrilir.

Bir derleyici, bir programlama dilinde yazılmış bir program için o programa eş değer olan makine dilinde bir program oluşturur.

DERLEME SÜRECİ

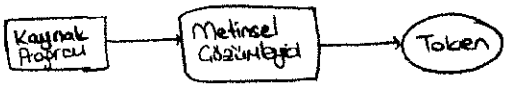


Derleyicinin Genel Çalışma şekli;
* Tüm programı bir kerede denetler.
* Sözlüksel ve sözdizimsel hataları bulur.
* Hata yok ise, programı nesne koda çevirir.
* Nesne kod daha sonra çalışabilir koda çevrilir



Metinsel (Lexical) Analiz

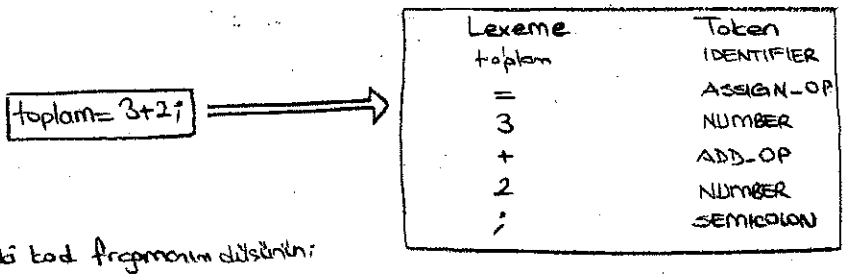
Bir derleyicinin ön ucunda bulunan metinsel çözümleyici, bir kaynak programı bir dizi token'a çevirir.



ÖRNEK: toplam := değer + 10;

toplam	(tanımlayıcı, gösterge1)
:	(iki_nokta, nil)
=	(eşit, nil)
değer	(tanımlayıcı, gösterge2)
+	(artı, nil)
10	(tamsayı, gösterge3)
;	(noktalıvirgül, nil)

} Token Dizisi



Aşağıdaki kod fragmanını düşünün:

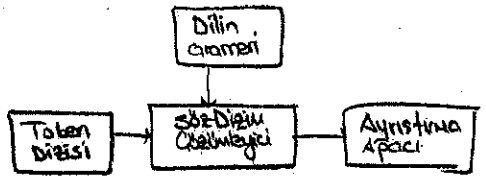
```
if (i==1);
    z=1;
else;
    z=0;
endif;
```

} Lexical analizci bunu karakterler stringi olarak okur.

i f _ (i = = 1) ; i n t z = 1 ; e l s e ; i n t z = 0 ; e n d i f ;

} Lexical analiz stringi tokenlara ayırır.

Sözdizim (Syntax) Çözümleme



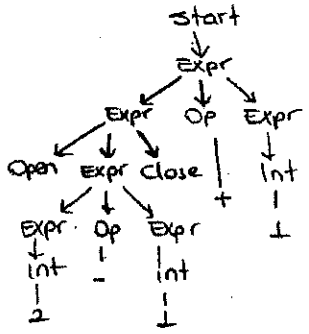
ve

(Ayrıştırma Ağacı)

Start → Expr
Expr → Expr Op Expr
Expr → Int
Expr → Open Expr Close

(2-1)+1 'i elde et.

Open / Close
Parantez açılıp kapatma.

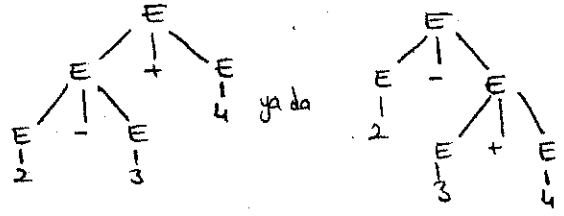


Belirsizlik

Aynı string için iki (ya da daha fazla) ayrıştırma ağacı

E → E+E
E → E-E
E → 0 | .. | 9

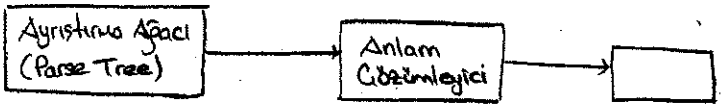
2-3+4



↓ Belirsiz bir gramer bazen belirli hale getirilebilir

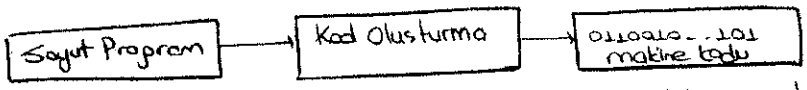
E → E+T | E-T | T
T → 0 | _ . | 9

Anlam (Semantics) Çözümleme



Soyut Dil ile Gösterilen Ara Program

Sayısal Uyg. • Algoritma çözümleme [illegible]
düzeyli bir birleştirici diline benzerler.



* Derleyicinin ön ucu programlama diline bağımlı, arka uçu ise bilgisayara bağımlıdır.

Optimizasyon (Eniyileme)

Optimizasyon ile bir programın daha etkin olarak çalışacak bir eş değer programa dönüştürülmesi amaçlanır. Programın daha hızlı çalışmasını, daha az bellek kaplamasını, kaynak kullanımında tutumlu davranmasını sağlar.

sonuc = fonksiyon1(a+b) + fonksiyon2(a+b) —optime→ c = a+b; sonuc = fonksiyon1(c) + fonksiyon2(c) } makine bağımsız

Eniyilestirilmemiş ⇒ carpım, ekle } makine bağımlı
Eniyilestirilmiş ⇒ shift, artış komut }

int x = 10+2;
int y;
y = x+5;
} Burada derleyici x içerisindeki ifadenin değişmediğini görerek y=x+5; yerine y=17; yerleştirebilir.

ÖRNEKLER

1)
```
inline int Kare (int a)
{return a*a;}
```
↓
```
inline int Kare (int a)
{ a = Kare (b);
  a = b*b; }
```

2)
```
for (;;) {
  x=10;
  /***/ }
```
↓
```
for (;;){
x= func();
 /***/
}
```

3)
```
for (i=0; i<strlen(s); i++) {
        /****/ }
```
↓
```
length = strlen(s);
for (i=0; i<length; i++){
    /****/
}
```

4)
```
for (i=0; i<100; i++)
   if (i%2 == 0)
       ifade1;
   else
       ifade2;
```
↓
```
for (i=0; i<100; i+=2)
     ifade1;
for (i=1; i<100; i+=2)
     ifade2;
```

5)
```
for (i=0; i<100; i++)
   Func(i);
```
↓
```
for (i=0; i<100; i+=5){
  Func(i);
  Func(i+1);
  Func(i+2);
  Func(i+3);
  Func(i+4);
}
```

Kod Üretimi Aşamasında Yapılan Optimizasyonlar

Derleyicinin hız optimizasyonunda en hızlı bir biçimde çalışacak komutları seçmek gerekir. Benzer biçimde büyüklük optimizasyonunda da derleyici en az yer kaplayacak makine komutlarıyla programının istediği işleri yapmaya çalışır. Genel olarak CISC tabanlı işlemcilerde komutlar farklı uzunluklarda olma eğilimindedir. Halbuki RISC işlemcilerinde eşit uzunluktadır.

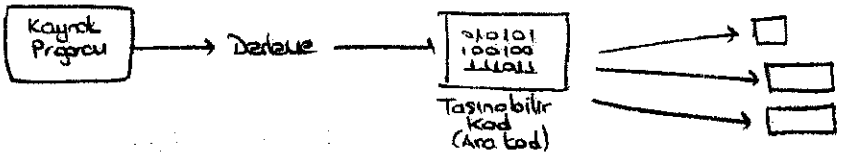
Komut seçimi en önemli problemlerden biridir. Bu aşama RISC işlemcilerinde daha kolay, CISC işlemcilerinde daha zor gerçekleştirilir. Çünkü RISC işlemcilerinde komut sayısı azdır ve aralarında ciddi bir performans farkı yoktur.

* Derleme yöntemi, yorumlamaya göre zamandan kazanç sağlar.
* Yorumlama, derleme yaklaşımına göre daha az bellek kullanır. Ama aday sırasında yorumlayıcı tüm alt programları bellekte tutuluyorsa yorumlayıcı bellek israfına neden olabilir.
* Yorumlayıcılar her komutu birer birer ele alır, hataların kullanıcıya bildirilmesi doğrudan gerçekleşir. Derleme kısmında ise tüm program bittikten sonra hatalar bildirildiği için güç olabilir. Yorumlayıcı hata bildirmede daha iyidir.

	Zaman	Bellek Kullanımı	Hata bildirme
Derleme	V	X	X
Yorumlama	X	V	V

Taşınabilir Kod

Bir program, kaynak program üzerinde basit düzenlemeler yaparak bir ortamda çevirilebilir. Bu özellik, ağırlıklı olarak derlemeye dayanır ve farklı makinelerde çalıştırılabilen taşınabilir kod üretmek amacıyla kullanılabilir.



Yazılım Geliştirme Yaşam Döngüsü

1) Gereksinimlerin Analizi ve Belirlenmesi
2) Sistem Tasarımı
3) Gerçekleştirim ve Birim Sınama
4) Sınama
5) Bakım

BÖLÜM-1

PROGRAMLAMA ALANLARI

* Bilimsel Uygulamalar * Ticari Uygulamalar * Yapay Zeka * Sistem Programlama * Web Yazılımı

Dil Değerlendirme Kriterleri

* Okunabilirlik (Readability) = Programın okunması, anlaşılması
* Yazılabilirlik (Writability) = Bir dilin program yazmadaki kolaylığı
* Güvenilirlik (Reliability) = Şartlara uygunluk
* Maliyet (Cost) = En son toplam maliyet

⇒ Bir programlama dilinin okunabilir ve anlaşılabilir olması, programda hata oranını azaltır ve programın bakımını kolaylaştırır.

* Genel Kolaylık: Aynı operasyonlar için farklı farklı özelliklerin bulunmaması genel kolaylığı arttırır.

```
count = count + 1
count += 1
count ++
++ count
```

Örneğin; (+) operatörü bir çok işlemde kullanıldığı için şüpheye düşürebilir. Ve bu okunabilirliği azaltır.

* Sonuç olarak: Basitlik okunabilirliği arttırır ama çok fazla basitlikte okunabilirlik uygunsuzlaşabilir. Assembly
* Ortogonallik: Bir dilde ortogonallik özelliğinin fazla olması, dildeki kodlamayı kolaylaştırır ve dilin öğrenmesini basit hale getirir.
* Kontrol İfadeleri
* Veri Tipleri ve Veri Yapıları: Esneklik, okunabilirliğe katkı sağlamaktadır.

```
Timeout = 1   yerine   Timeout = true
```

yazmak okunabilirliği arttırır.

* Syntax Tasarımı: Çok önemli bir etken (okunabilirlik için)
  - Anlaşılır yapılar kullanılmalı
  - Karmaşık yapılar için özel metotlar kullanılmalı
  - Anlamlı anahtar sözcükler kullanılmalı

⇒ Programlama ağının ne kadar kolay okunduğunun okunmasına yazılabilirlik denir. Okunabilirliği etkileyen faktörler yazılabilirliği de etkilemektedir.

- Basitlik ve Ortogonallik : Az sayıda ilkel yapı ve bunları birleştirmek için oluşturulan legal kombinasyonlar (ortogonallik) kullanmak yazılabilirliği arttırmaktadır. Çok fazla ortogonallik; okunabilirliği azalttığı gibi yazılabilirliği de azaltabilmektedir. Fazla ortogonallik programdaki hataları bulmayı zorlaştırmaktadır.
- Soyutlama (Abstraction) : Karmaşık yapıları veya işlemleri, ayrıntıları göz ardı ederek tanımlayabilme ve kullanabilme yeteneğine soyutlama denir. Programlama dilinde kullanılan soyutlamanın derecesi ve doğal ifade edebilme özellikleri yazılabilirliği etkilemektedir.
- Anlamlılık (Expressivity) : İşlemleri belirtmenin nispeten elverişli yollarını kullanmak yazılabilirliği arttırır. count++ ifadesi count = count + 1 ifadesinden daha açıklayıcıdır. Yazılabilirliği arttırır.

⇒ Bir program tüm şartlar altında istenilen görevi doğru olarak yapıyorsa güvenilirdir denir.
Bir programlama dilinin güvenilirliği, o dil kullanılarak geliştirilen programların güvenilir olmasıdır.
* Tip kontrolü, istisna işleme, farklı adlandırma, okunabilirlik ve yazılabilirlik

— Tip Kontrolü : Programın tip hatalarının test edilmesi. Derleme zamanında az maliyetli, çalışma zamanında çok maliyetlidir tercih edilmez.

— İstisna İşleme : Yürütme zamanı (run-time) hatalarını yakalayıp ve düzeltme önlemleri alır.

- Farklı Adlandırma : Aynı hafıza hücresine erişim için 2 veya daha fazla farklı ismin tanımlanmasıdır.

⇒ Maliyet
* Dili kullanacak programcıların eğitilmesi
* Programları yazma
* Güvenilirlik
* Programları derleme
* Programları yürütme
* Bakım

Dil Tasarımını Etkileyen Faktörler

Bilgisayar Mimarisi

Diller, von Neuman mimarisi olarak bilinen yaygın bir bilgisayar mimarisi etrafında geliştirilir.
Von Neumann mimarisinde : Veri ve programlar bellekte saklanır
Bellek CPU'dan ayrıdır
Komutlar ve veri bellekten CPU'ya iletilir.
CPU'daki işlemin sonucu hafızaya geri gönderilir

Programlama Paradigmaları

Bir paradigma, bir grubun ortak kararlarını temsil eden ve grubun konuya yeni bir şekilde bakışını sağlayan bir kavramsal şemadır.

İmprative	Fonksiyonel	Mantık	Nesneye Yönelik
↓	↓	↓	↓
Komut ve işlemlerin sırası belirtilir (Atama ifadeleri değişkenler)	Fonksiyonlar üzerine kurulmuş bir model	Kural tabanlı Ayrıntı yok	Veri soyutlama Kalıtım bağlama

İmplementasyon Metotları

* Derleme (Compilation) : Programlar makine diline çevrilir.
* Saf Yorumlama (Pure Interpretation) : Programlar yorumlayıcı olarak bilinen başka bir program tarafından yorumlanır.
* Hibrit : Derleyiciler ve saf yorumcular arasındaki uzlaşmadır. Saf yorumlamadan daha hızlıdır.

Von Neumann Darboğazı (Bottleneck)

Bir bilgisayarın belleği ve işlemcisi arasındaki bağlantı hızı o bilgisayarın hızını belirler. Çoğu kez program komutları bu hızdan daha hızlı yürütülebilir, bağlantı hızı bu yüzden bir darboğaza sebep olur.
Bilgisayarların hızını sınırlayan birinci faktördür.

BÖLÜM-2

Java Değerlendirmesi

* C++'ın güvensiz özelliklerini elemiştir.
* Eş zamanlılık özellikleri
* Applettler için kitaplıklar, GUI'ler, veritabanı erişimi
* C++ tabanlı olarak geliştirilmiştir.
* Basit, taşınabilir ve nesn
* Bir Java programının temel yapısal bileşeni sınıftır. Bütün veri ve metotlar bir sınıf ile ilişkilidir.
* C++'ta bulunan çoklu miras alma, operatörlerin üst üste bindirilmesi, ve makro önişlemcisi özellikleri Java'da yoktur.
* Tip dönüşümü kuvvetli.
* Java internet ve web programcılığında yaygın olarak kullanılmaktadır.

BÖLÜM-3 (Sentaks ve Semantiği Tanımlama)

Sözdizim ⇒ Sentaks Anlam bilimi ⇒ Semantik

*Deyimin sonuna noktalı virgül konması sentaks kuralı, değişkenin kullanılmadan önce tanımlanması semantik kuralı örneğidir.

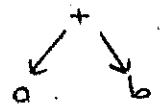
- Bir dil tanımının kullanıcıları
  * Diğer dil tasarımcıları
  * Uygulamacılar
  * Programcılar

Sözdizim (Syntax) ⇒ Bir dilin sözdizim kuralları, bir deyimdeki her kelimenin nasıl yazılabileceğini belirler.

Anlam (Semantics) ⇒ Bir dilin anlam kuralları ise, bir program çalıştırıldığında gerçekleşecek işlemleri tanımlar.

Soyut Sözdizim

+ab	prefix
a+b	infix
ab+	postfix



Metinsel Sözdizim

puan = 4 * dogru + 10 ;

Lexeme	Token
puan	Tanımlayıcı
dogru	
4	tamsayı_sabit
10	
=	eşit_işareti
*	carpım_işlemcisi
+	toplama_işlemcisi
;	noktalı virgül

DİLLERİN FORMAL TANITIMLARI

Dil Tanıyıcılar

Verilen bir programın bir dilde olup olmadığına karar veren bir cihaz (otomatlar)

Dil Üreteçler

Bir dilin cümlelerini üretmek için kullanılabilen cihaz (context-free)

* Bir ya da daha çok dilin sözdizimini anlatmak amacıyla kullanılan dile metadil adı verilir.

SENTAKS TANIMLAMANIN BİÇİMSEL METOTLARI

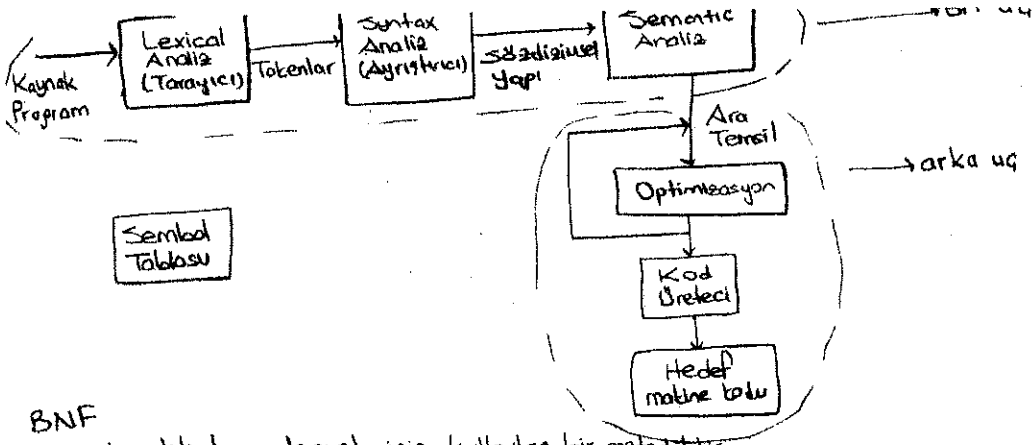
BNF ve içerik bağımsız Context Free Gramerler

- Programlama dil sentaksını tanımlayan en çok bilinen metottur.

(Genişletilmiş) Extended BNF

- BNF'in okunabilirliğini ve yazılabilirliğini arttırır.

Sözdizimi → GRAMER (Kurallar Bütünü)



BNF

Başka bir dili tanımlamak için kullanılan bir metadildir.

BNF'de açıklanan bir gramer ; 1) Terminal semboller : Daha alt parçalara ayrılamayan semboller (+, *, -, if)

2) Terminal olmaya " : "<" ">" sembolleri arasında gösterilir.

3) Kurallar : Kurallar terminal olmayan sembol (→, ::=) T.S, T.O.S
Kurallar "|" bu sembolle ayrılır.

4) Başlangıç sembolü : Dilin ana elemanını göstermek için, (amaç sembol)

* Bir türetme, başlangıç sembolüyle başlayan ve bir cümleyle biten kuralların tekrarlı bir uygulamasıdır.

ÖRNEK

Gramer Örneği

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

Türetme →

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

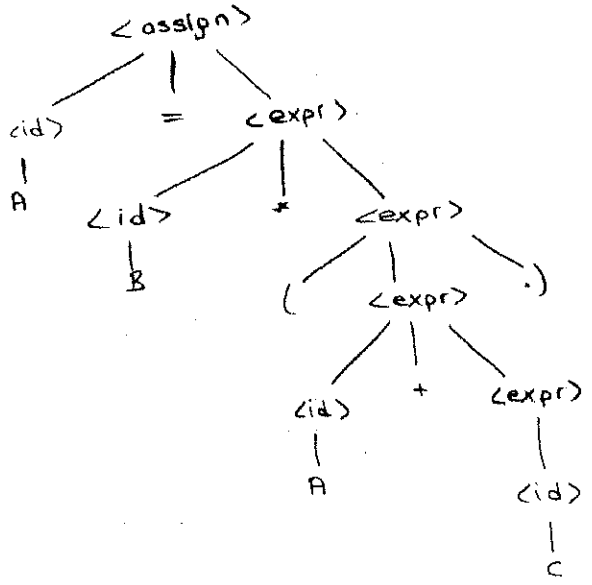
ÖRNEK

Gramer

<assign> → <id> = <expr>
<id> → a | b | c
<expr> → <id> + <expr> | <id> * <expr>
| (<expr>) | id

Sdo dayalı türetme ile a = b * (a + c)'yi türet.

<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)

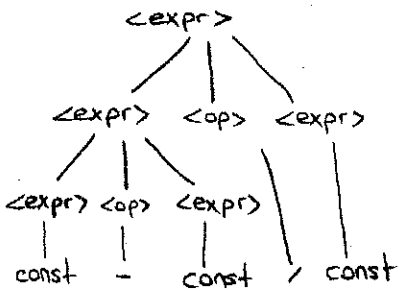


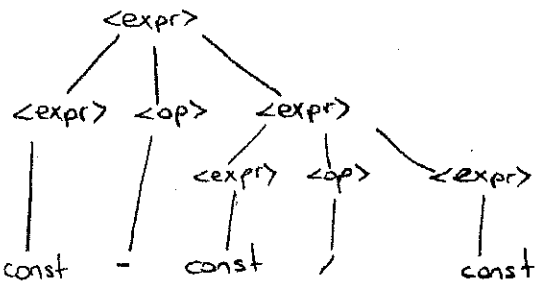
* Bir gramer ancak ve ancak iki veya daha fazla farklı ayrıştırma ağacı olan bir cümlesel kısım üretiyorsa belirsizdir.

Belirsiz Deyim Gramerı

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

const - const / const, 'u elde et.



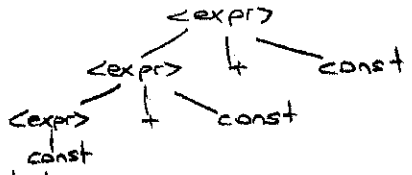


* Eğer ayrıştırma ağacını operatörlerin öncelik seviyelerini göstermek için kullanırsak, belirsizlik olmaz.

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

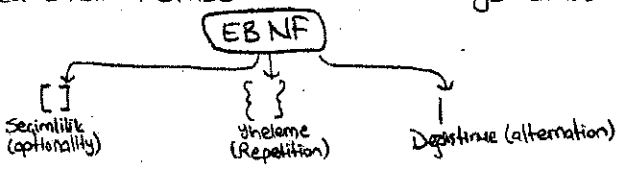
* Operatör birleştirilişi de gramerle gösterilebilir.

<expr> → <expr> + <expr> | const (Belirsiz)
<expr> → <expr> + const | const (Belirli, kesin)



EBNF (Extended BNF)

Seçimlilik (optionality), yineleme (repetition) ve değiştirme (alternation) olmak üzere üç özellik yer almaktadır.



Seçimlilik [] (optionality)

[ ] içindeki bölüm, bir kural tanımında hiç yer almayabilir veya bir kez bulunabilir.

EBNF <seçimlilik_deyim> ⟶ if (<mantiksal>) <deyim> [else <deyim>];

BNF <seçimlilik-deyim> ⟶ if (<mantiksal>) <deyim>
<seçimlilik_deyim> ⟶ if (<mantiksal>) <deyim> else <deyim>;

[ ] kullanılmadığı durumda, bu if deyiminin yukarıda gösterildiği gibi iki kural ile açıklanması gereklidir.

Yineleme { } (repetition)

Bir kuralın sağ tarafında, istenilen sayıda yinelenebilecek veya hiç yer almayabilecek bir bölümü göstermek için { } kullanımı eklenmiştir. EBNF'teki yineleme sembolü ile, BNF'teki iki kural olarak gösterilen tanımlamalar, tek kural ile ifade edilebilmektedir.

<tanımlayıcı_listesi> ⟶ <tanımlayıcı> { <tanımlayıcı> }

<tanımlayıcı> ≡ <tanımlayıcı> <tanımlayıcı> <tanımlayıcı> - - -

Deyimler (cont.)

Bir grup icinden tek bir eleman secilmesi gerektigi zaman secenekler, parantezler icinde birbirlerinden "veya" islemesi "|" ile ayrilarak yazilabilir.

EBNF <for_deyimi> → for <degisken> := <ifade> (to | downto) <ifade> do <deyim>

BNF <for_deyimi> → for <degisken> := <ifade> (to) <ifade> do <deyim>
<for_deyimi> → for <degisken> := <ifade> (downto) <ifade> do <deyim>

ÖRNEK

<amac> := [x] y {z}

y ⟶ x secilmedi, z yinelenmedi.
x y ⟶ x secildi, " "
y z ⟶ x secilmedi, z bir defa yinelendi.
x y z ⟶ x secildi, " " "
y z z ⟶ x secilmedi, z iki defa yinelendi.
x y z z ⟶ x secildi, z iki defa "
y z z z z ⟶ x secilmedi, z dört defa yinelendi.

ÖRNEK

BNF

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

EBNF

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

Özellik Grameri / Örnek

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

- actual_type : <var> ve <expr> ile sentezlenmistir
- expected_type : <expr> ile miras birakilmistir.

Programlama Dillerinin Anlamsal (Semantik) Olarak Tanimlanmasi

Duragan (Statik) Anlam Kurallari

Bazi kurallarin BNF ile gösterilmesi olasi degildir. Örnegin; degiskenlerin kullanilmadan önce tanimlanmasi gibi. Bu ve benzeri kurallar, dilin duragan anlam kurallaridir.

Dinamik Anlam Kurallari

Bir dildeki çesitli yapilarin calistirilmasinin etkilerini aciklayan kurallara, dilin dinamik anlam kurallari denir.

RESMİ ANLAM TANIMLAMA

a) İşlemsel (Operational) Semantik

Bir programi simülasyon veya gerçek olarak makine üzerinde calistirarak anlamini aciklamaktir. İşlemsel anlamlarda, bir ifadenin ya da programin anlamini daha iyi anlamak için daha kolay anlasilir bir dile çevirme islemi vardir.

C ifadesi

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

Gerçeklestirme kolay,
Muhakeme zor

İşlemsel anlam

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

b) Kurala Dayalı (Axiomatic) Semantik
Biçimsel mantığa dayalıdır.

$a = b + 1 \quad \{a > 1\}$

Mümkün bir ön şart : $\{b > 10\}$
En zayıf ön şart : $\{b > 0\}$

$\{x > 5\}\ x := x - 1\ \{x > 0\}$

$x - 1 > 0$
$x > 1$ en zayıf ön koşul $x > 1$ burada $\{x > 5\}$ ön koşulunu da imo eder yani doğruluk kanıtlıyor.

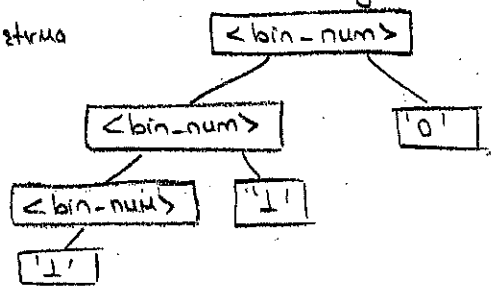
$x = 2 * y - 3 \quad \{x > 25\}$ son koşul.
$2 * y - 3 > 25$
$y > 14$ olması gerekir.
En zayıf ön şart $\{y > 14\}$ dir.

⇒ İspatların doğruluğu için iyi bir araçtır fakat dil kullanıcı için kullanışlı değildir.

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.

c) Fonksiyonel (Denotasyonel) Semantik
Özyinelemeli (recursive) fonksiyon teorisine dayalıdır. (En soyut) En yaygın semantik tanımlama metodudur.

1 1 0 için ayrıştırma ağacı



⇒ Programların doğruluğunu ispatlama için kullanılabilir.
Dil tasarımında yardımcı olabilir.
Karmaşık olduğu için dil kullanıcıları tarafından az kullanılmıştır.

İşlemsel semantikte ⇒ Kodlanmış algoritmalar
Denotasyonel " ⇒ Sıkı matematiksel fonksiyonlar

Sözcüksel (lexical) ve sentaks (syntax) analizini ayırmanın nedenleri:

Basitlik: Sözcüksel analiz için daha az karmaşık yaklaşımlar kullanılabilir; bunları ayırmak ayrıştırıcıyı basitleştirir.

Verimlilik: Ayırmak sözcüksel analizcinin optimizasyonuna imkan verir, sentaks analizciyi optimize etmek sonuç vermez.

Taşınabilirlik: Sözcüksel analizcinin bölümleri taşınabilir olmayabilir, fakat ayrıştırıcı her zaman taşınabilirdir.

Sentaksı tanımlamak için BNF kullanmanın nedenleri:

* Net ve özlü bir sentaks tanımı sağlar.
* Ayrıştırıcı doğrudan BNF ye dayalı olabilir.
* BNF ye dayalı ayrıştırıcıların bakımı daha kolaydır.

Sözcüksel (Lexical) Analiz

```
program gcd (input, output);
var i,j : integer;
begin
    read (i,j);
    while i<>j do
    if i>j then i:=i-j else j:=j-i;
    writeln(i)
    end
```

⇓

program	gcd	(	input	,	output	)	;	var	i	,	j	:	integer	;						
begin	read	(	i	,	j	)	;	while	i	<>	j	do	if	i	>	j	then	i	:=	i
-	j	else	j	:=	i	-	i	;												
writeln	(	i	)	end	.															

(( )) ( ) için Leftmost Türetme

Gramer

L → (L) L

L → ε

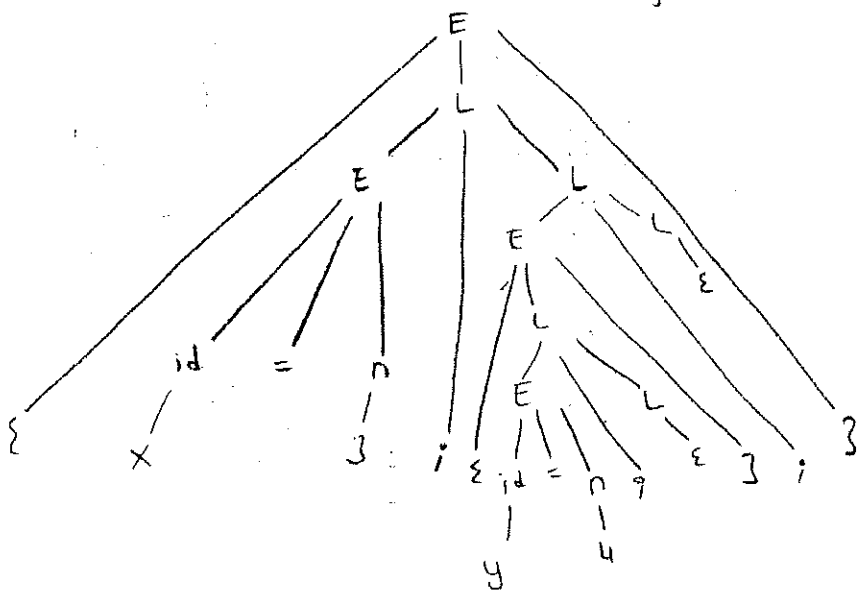
L → (L) L
→ ((L) L) L
→ (( ) L) L
→ (( )) L
→ (( )) (L) L
→ (( )) ( ) L
→ (( )) ( )

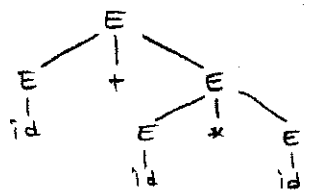
Yukarıdan Aşağıya Ayrıştırıcı

E → id = n | { L }

L → E ; L | ε

{ x = 3 ; { y = 4 ; } ; } için ayrıştırma ağacını göster.





E → E+E
→ id+E
→ id+E*E
→ id+id*E
→ id+id*id

Pairwise Disjointness Testi

Her bir nontermind A için, birden fazla sağ kısmı olan grammerde, her bir kural çifti $A \to a_i$ ve $A \to a_j$ için, şu doğru olmalıdır:

$$FIRST(a_i) \cap FIRST(a_j) = \phi$$

Türetmede üretilecek ilk terminal sembol tek olmalıdır.

$A \to a \mid bB \mid cAb$ ⟶ Disjoint

$A \to a \mid aB$ ⟶ disjoint değil !

ÖRNEK

(n+(n).) * n $

E → TX
X → ATX | λ
A → + | -
T → FN
N → MFN | λ
M → *
F → (E) | n

ÇÖZÜM

E → TX
→ FNX
→ (E)NX
→ (TX)NX
→ (FNX)NX
→ (nNX)NX
→ (nX)NX
→ (nATX.)NX
→ (n+TX)NX
→ (n+FNX).NX
→ (n+(E)NX)NX
→ (n+(TX)NX)NX
→ (n+(FNX)NX)NX
→ (n+(nNx)Nx)Nx
→ (n+(nX.)NX)NX
→ (n+(n)NX)NX
→ (n+(n)X)NX
→ (n+(n))NX
→ (n+(n))MFNX
→ (n+(n))*FNX
→ (n+(n)) * nNX
→ (n+(n)) * nX
→ (n+(n))*n

First Kümesi

ÖRNEK

exp → exp addop term | term
addop → + | -
term → term mulop factor | factor
mulop → *
factor → (exp) | num

First(addop) = {+, -}
First(mulop) = {*}
First(factor) = {(, num}
First(term) = {(, num}
First(exp) = {(, num}

ÖRNEK

st → ifst | other
ifst → if (exp) st elsepart
elsepart → else st | λ
exp → 0 | 1

First(exp) = {0, 1}
First(elsepart) = {else, λ}
First(ifst) = {if}
First(st) = {if, other}

Follow Kurallar

exp → term exp'

exp' → addop term exp' | λ

addop → + | -

term → factor term'

term' → mulop factor term' | λ

mulop → *

factor → (exp) | num

	First	Follow
exp	(, num	$)
exp'	+, -, λ	$)
addop	+, -	(, num
term	(, num	+, -, $,)
term'	*, λ	+, -, $,)
mulop	*	(, num
factor	(, num	*, +, -, $,)

LL(1) Ayrıştırma Tablolarını Oluşturma

1) exp → term exp'
2) exp' → addop term exp'
3) exp' → λ
4) addop → +
5) addop → -
6) term → factor term'
7) term' → mulop factor term'
8) term' → λ
9) mulop → *
10) factor → (exp)
11) factor → num

	(	)	+	-	*	num	$
exp	1					1	
exp'		3	2	2			3
addop			4	5			
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	

- First'lere göre yapılıp, λ'larda follow'lara bakılıyor.

→ En fazla bir kuralı sahipse LL(1) gramerdir.

LL(1) olmayan Gramer için LL(1) Ayrıştırma Tablosu

1) exp → exp addop term
2) exp → term
3) term → term mulop factor
4) term → factor
5) factor → (exp)
6) factor → num
7) addop → +
8) addop → -
9) mulop → *

	(	)	+	-	*	num	$
exp	1,2					1, 2	
term	3,4					3,4	
factor	5					6	
addop			7	8			
mulop					9		

First(exp) = { (, num }

First(term) = { (, num }

First(factor) = { (, num }

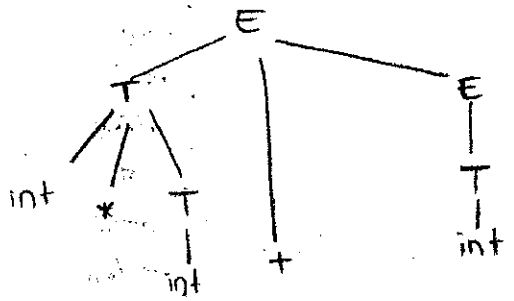
First(addop) = { +, - }

First(mulop) = { * }

Grameri LL(1) yapmayan nelerdir? → sol özyineleme, sol faktör

Aşağıdan-Yukarıya Ayrıştırma

int * int + int
int * T + int
T + int
T + T
E

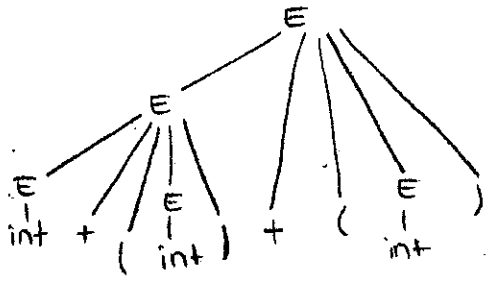


ÖRNEK

$$\begin{bmatrix} E \to int \\ E \to E + (E) \end{bmatrix}$$ Gramer

int + (int) + (int)
E'yi elde et.

int + (int) + (int)
E + (int) + (int)
E + (E) + (int)
E + (int)
E + (E)
E



SHIFT-REDUCE

ÖRNEK

$$\begin{bmatrix} E \to int \\ E \to E + (E) \end{bmatrix} \Rightarrow$$ Gramer

E'yi elde et.

Başlangıç:

↑int + (int) + (int) $	shift
int↑ + (int) + (int) $	reduce E→int
E↑ + (int) + (int) $	shift 3 kez
E + (int↑) + (int) $	reduce E→int
E + (E↑) + (int) $	shift
E + (E) ↑ + (int) $	reduce E→E+(E)
E↑ + (int) $	shift 3 kez
E + (int↑) $	reduce E→int
E + (E↑) $	shift
E + (E) ↑ $	reduce E→E+(E)
E↑ $	

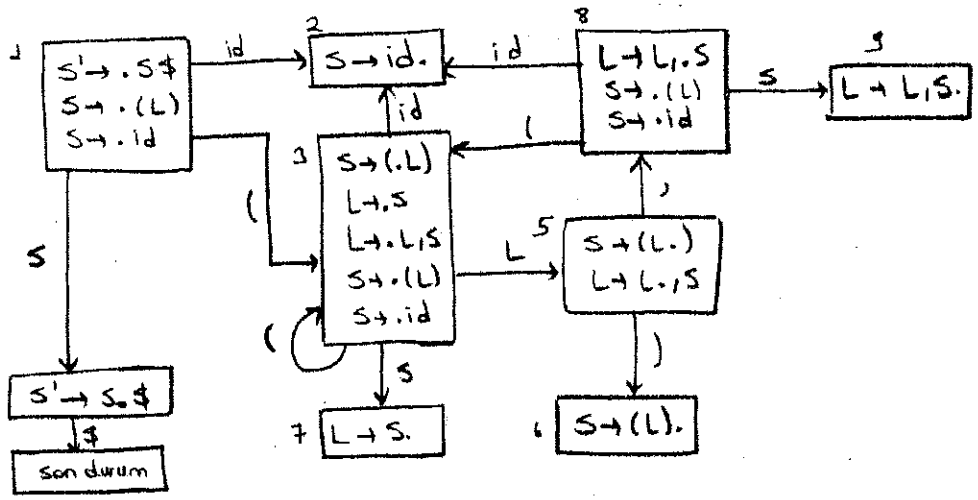
ÖRNEK

S → aABe
A → Abc | b
B → d

Stack	Input	Action
$	abbcde$	Shift
$a	bbcde$	shift
$ab	bcde$	Reduce A→b
$aA	bcde$	Shift
$aAb	cde$	Shift
$aAbc	de$	Reduce A→Abc
$aA	de$	shift
$aAd	e$	Reduce B→d
$aAB	e$	shift
$aABe	$	Reduce S→aABe
$	$	End

TAM DFA

Gramer
S → (L) | id
L → S | L,S



Ayrıştırma Örneği

S → (L) | id
L → S | L,S

türetme	yığın	giriş	hareket
((x),y)		((x),y)	shift, goto3
((x),y)	(	(x),y)	shift, goto3
((x),y)	((	x),y)	shift, goto2
((x),y)	((x	),y)	reduce S→id
((S),y)	((S	),y)	reduce L→S
((L),y)	((L	),y)	shift, goto6
((L),y)	((L)	,y	reduce S→(L)
(S,y)	(S	,y)	reduce L→S
(L,y)	(L	,y)	shift, goto8
(L,y)	(L,	y)	shift, goto9
(L,y)	(L,y	)	reduce S→id
(L,S)	(L,S	)	reduce L→L,S
(L)	(L	)	shift, goto6
(L)	(L)		reduce S→(L)
S	S	$	done

Liste Gramer Ayrıştırma Tablosu

	(	)	id	,	$	S	L
1	S3		S2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	S3		S2			g7	g5
4					accept		
5		S6		S8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	S3		S2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

Shift-Reduce Ayrıştırma Örneği

Gramer: $S' \rightarrow S$
$S \rightarrow (S)S \mid \lambda$

Yığın	Giriş	Hareket
$	(())$	shift
$(	())$	shift
$((	))$	reduce S→λ
$((S	))$	shift
$((S)	)$	reduce S→λ
$((S)S	)$	reduce S→(S)S
$(S	)$	shift
$(S)	$	reduce S→λ
$(S)S	$	reduce S→(S)S
$S	$	accept

YIĞIN	GİRİŞ TAMPONU	ACTION
$	num1+num2*num3$	shift
$num1	+num2*num3$	reduce
$F	+num2*num3$	reduce
$T	+num2*num3$	reduce
$E	+num2*num3$	shift
$E+	num2*num3$	shift
$E+num2	*num3$	reduce
$E+F	*num3$	reduce
$E+T	*num3$	shift
$E+T*	num3$	shift
$E+T*num3	$	reduce
$E+T*F	$	reduce
$E+T	$	reduce
$E	$	accept

ÖRNEK

Gramer

E → E+T | T | E-T

T → T*F | F | T/F

F → (E) | id | -E | num

Cevap yan tarafta.

*Amaç başlangıç sembolü ile sonuçlandırmak.

DEVAM--

Imperative programlama, programların işlem deyimleri ile bellekteki değerleri değiştirmesine dayanır.

isim: Programdaki diğer deyimler tarafından o değişkene başvuru için kullanılır.
adres: Bir değişken bir program içerisinde farklı yerlerde farklı adreslere sahip olabilir.
değer: Bir değişkenin değeri, belirli bir yönteme göre kodlanmış olarak saklanır.
tip: Bir değişkenin tipi, o tip için belirlenmiş işlemleri belirler.
yaşam süresi: Bir değişkenin yaşam süresi, bir bellek adresinin bir değişken ile ilişkili kaldığı süredir.
kapsam: Bir değişkenin kapsamı, o değişkenin isminin tanımlı olduğu program deyimlerini göstermektedir.

Fonksiyonel Yan Etkiler

* Bir fonksiyon iki yönlü bir parametreyi veya lokal olmayan bir değişkeni değiştirdiğinde meydana gelir.

ÖRNEK

Bir ifadede çağrılmış bir fonksiyon ifadenin başka bir operantını değiştirdiğinde ortaya çıkar; bir parametre değişim örneği:

```
int fun (int *u) {
        *u = *u/2;
        return *u; }

a=10;
b=a+ fun(&a); /* fun, parametresini değiştiriyor */
```

Bu problem için 2 muhtemel çözüm:

1) Fonksiyonel yan etkileri iptal etmek için dil tanımlaması yapılır.
   (2 yönlü parametre olmayacak, global değişken olmayacak)
2) Operantların işlem sırasını belirlemek için dil tanımlaması yapılır.

Kısa Devre Tespiti

```
(13*a) * (b/13-1)
(a>=0) && (b<10)
```

Bir ifadede operant / operatörlerin tüm hesaplamalarını yapmaksızın sonucun bulunmasıdır.

Eğer a=0 ise diğer kısmı hesaplamaya gerek yoktur. (b/13-1)

Kısa Devre Olmayan

```
index=0;
while (index <=length) && (LIST [index] != value)
        index ++;
```

• index = length olduğunda, LIST [index] indetsleme problemi ortaya çıkaracak (LIST dizisi length-1 uzunluğunda varsayılmış)

BAĞLAMA (BINDING) KAVRAMI

Bir program elemanı ile bir özellik arasında ilişki kurulmasına bağlama (binding) denir.

Durağan (Statik) Bağlama

1) Dil Tasarım Zamanında Bağlama 2) Dil Gerçekleştirim Zamanında Bağlama 3) Derleme Zamanında Bağlama

Dinamik Bağlama

4) Çalışma Zamanında Bağlama

ÖRNEK	Dil Tasarımı Zamanında Bağlama	Dil Gerçekleştirim Zamanında Bağlama	Derleme Zamanında Bağlama	Çalışma Zamanında Bağlama
int puan; puan = puan + 3;	- puan değişkeninin alabileceği tipler - + sembolünün alabileceği anlamlar	- puan değişkeninin alabileceği değerler	- puan değişkeninin programdaki tipi - + sembolünün anlamı	- puan değişkeni'nin o değişkendeki değeri

TİP BAĞLAMA

Tip bilgisi, bir tanımlayıcı ile ilişkilendirilince, tanımlayıcı, o tiple bağlanmış olur. Örneğin; Birçok programlama dilinde bir programda bir değişkene başvuru yapılmadan önce, değişkenin bir veri tipi ile bağlanması gerekir.

Derleme Zamanı	Çalışma Zamanı
STATİK TİP BAĞLAMALI DİLLER	DİNAMİK TİP BAĞLAMALI DİLLER

Durağan (Statik) Tip Bağlama

Tiplerin isimlerle derleme zamanında bağlandığı diller olarak nitelendirilirler. Burada derleyici, tip hatalarını, program çalıştırılmadan önce yakalar.

Örtülü Tanımlama

PL/I'da dışsal olarak tanımlama dışı durumlarda, isimleri "I" ve "N" arasındaki harflerle başlayan değişkenler tamsayı tipi ile bağlanır.

Örtülü tanımlamalarda programda farkedilmesi güç hatalara yol açtığı için ve programlama dili güvenilirliği olmadığı için günümüzde çoğu programlama dili, değişkenlerin dışsal olarak tanımlanması gerektirmektedir.

Dışsal Açık Tanımlama

İnteger tipi toplam değişkeni ile bağlanmıştır.

```
int toplam;
```

Dinamik Tip Bağlama

Bir programlama dilinde bir değişkenin tipi çalışma zamanında, değişkenin bağlandığı değer ile belirleniyorsa, dil dinamik tip bağlamalı olarak nitelendirilir.

→ Genellikle yorumlayıcı ile gerçekleştirilirler.

* Dinamik tip bağlamalı dillerde, dillerin derleyicilerinin hata yakalama yeteneği, durağan tip bağlamalı bir dilin derleyicisine göre zayıftır.
* Dinamik tip bağlama, programlarda durağan olarak tip denetimi yapılmasını önler.

BELLEK BAĞLAMA

- Bir değişkene bağlanan bir bellek hücresi, kullanılabilir bellek hücreleri arasından seçilir ve bu işleme bellek yeri atanması denir.
- Bir değişkenin kullandığı bellek hücresini geri vermesi ise belleğin serbest bırakması işlemi olarak nitelendirilir.

Bellek Düzeni

* Derlenmiş Program ⟶ Kodların tutulduğu bölüm
* Genel Değişkenler ⟶ Programdaki global değişkenleri içerir.
* Yığıt Bellek ⟶ Her alt program için etkinlik kaydı
* Yığın Bellek ⟶ Dinamik bellek değerleri için kullanılır.

Derleme zamanı	Çalışma zamanı
Durağan	Yığıt Dinamik
Dışsal yığın Dinamik	Örtülü Dinamik

alt program desteği.

Yığıt (Dinamik) Değişkenler: Özyinelemeye izin verir, depolamayı korur, az bellek harcamasına neden olur. / Alt prog. geçmişe hassas değildir. Çıkılınca tüm bilgiler unutulur.

Dışsal (Açık) Yığın Dinamik Değişkenler: Bellek gereksinimi görülmez ve veriler çalışma zamanında gerek oldukça belleğe atanır. Dışsal yığın dinamik değişkenlerin tip bağlaması derleme zamanında, bellek yeri bağlaması ise çalışma zamanında gerçekleşir.

`int *sakla;` ⟶ sakla isimli göstergesi değişken tanımlıyor

`sakla = new int;` ⟶ Sakla isimli dışsal yığın dinamik değişken yaratılıyor (new yöneticisi tip ismini kullanır)

`delete sakla;` ⟶ Değişken için ayrılan bellek geri veriliyor.

Örtülü Dinamik Değişkenler:
Sadece değer aldıkları zaman yığın belleğe bağlanırlar. Bu tür değişkenlerin tip ve bellek özellikleri her değer alışında yenilenebilir. (Esnektir)
Hata farketme yetersizliği vardır

Bağlaçlı listeler ve ağaçlar gibi çalışma sırasında büyüyebilen veya küçülebilen yapılar için uygun değişkenlerdir. Yönetimi zor, bu nedenle güvenilir değil.

DERLEME ZAMANI	ÇALIŞMA ZAMANI
Durağan (Statik) Değişkenler	**Yığıt Dinamik Değişkenler**
Bellek yeri, programın çalışması başlamadan bağlanır.	Bellek yeri, çalışma zamanında bellektedir, yığıt bellekten ayrılır
Dışsal Yığın Dinamik Değişkenler	
Veriler çalışma zamanında gerek oldukça belleğe atanır.	
Örtülü Dinamik Değişkenler	
Sadece bir değer aldıkları zaman yığın belleğe atanırlar.	

İSİM KAPSAMLARI

Durağan (Statik) Bağlam Kapsamı

Bu yöntemde değişkenlerin kapsamları, programın metinsel düzenine göre belirlenir.

Dinamik Kapsam Bağlama

Bir ismin kapsamının / altprogramların fiziksel yakınlıklarına göre değil; altprogramların çağrılma sırasına göre çalışma zamanında belirlenmesi dinamik kapsam bağlama olarak adlandırılır
- Değişkenlere çağrılma sırasında ters yönde arama yapılarak başvurulur.
- Bu durumda bir isim tanımı, çalışma sırasında aynı isimde yeni bir tanımlama bulunana kadar, kendisinden sonra çalıştırılan tüm komutlarda geçerlidir.

```
int x = 10;
void f1();
void main() {
  int x = 20;
  f1();
  cout << x;
}
void f1() {
  cout << x;
}
```

Çıktı nedir?
f1 hangi x?

main x 20
Global x 10

ÖRNEK

```
program L;
var n : char;
procedure W;
begin
writeln (n)
end;
procedwe D;
var n : char;
begin
n := "D";
W;
end;
begin {L}
n := "L"
W;
D
end.
```

→ char tipinde bir n tanımlanmış

→ Ana program kodları

Statik kapsam bağlama kuralına göre;

L
L

→ Programın ana bloğunda n kaçsa onu yazar. Çünkü n tanımlanmamış.

→ Prosedürleri çalıştırmadan ana program kodlarına gider.

→ Burda n yeniden tanımlanmış. Statik kapsamda bir üst bloğa bakılır.

Dinamik kapsam bağlama kuralına göre

L
D

* Burda ise D'de tanımlanmış.

* Çağrıldığı bloğa bakılır. Orda yoksa onu çağıran bloğa bakılır.

ÖRNEK

```
int x;
int main() {
x=2;
f();
g();
}
void f() {
int x=3;
h();
}
void g() {
int x=4;
h();
}
void h() {
printf("%d\n", x);
}
```

Statik Kapsam

main	f	g
x=2	x=3	4

→ 2
→ 2

Dinamik Kapsam

main	f	g
2	3	4

→ 3
→ 4

ÖRNEK

```
program MAIN;
var a : integer;
procedure P1;
begin
print a;
end ; {of P1}
procedure P2 ;
var a : integer;
begin
a := 0 ;
P1 ;
end; {of P2}
begin
a := 7;
P2 ;
end {of MAIN}
```

Statik Kapsam

main	P1	P2
a=7	print	a=0

→ 7

Dinamik Kapsam

main	P1	P2
a=7	print	a=0

→ 0

Tip Kontrolü

* İşlenenlerin işletmenlere uygunluğuna bakmak tip kontrolü olarak adlandırılır.
* İşlenen işletmene uygun değilse tip hatası oluşur.
* Eğer tip bağlanmaları statikse, tip kontrolü de statik.
* Eğer tip bağlanmaları dinamikse, tip kontrolü de dinamik

▽ Bir programlama dili eğer tip hatalarının hepsini fark ediyorsa bu dile kesin tiplendirilmiş dil denir. (strongly typed)

↳ Değişkenlerin yanlış kullanımının fark edilmesini ve hatalı sonuçları engeller.

Örneğin; JAVA'da bir tam sayı değişkeni ile gerçel sayı değişkeni toplanırken tam sayı otomatik gerçele çevirilir ve bu yapılırken kesin tipleme bozulmuş olur.

Referans Çevreleri

* Bir ifadenin referans çevresi ifadede görünen bütün isimlerin koleksiyonudur.
* Bir statik kapsamlı dil, lokal değişkenler artı bütün çevreleyen kapsamlardaki görünür değişkenlerin tümüdür.
* Bir alt programın çalıştırılması başlamış, ama henüz bitmemişse o alt program aktiftir.
* Bir dinamik kapsamlı dilde, referans platformu lokal değişkenler artı tüm aktif altprogramlardaki bütün görünür değişkenlerdir.

ÖRNEK

```
program A
x,y,z;
produce sub1
int y,z;
y=0 , z=0;
y=x+z;
z=y+20;
x=x+y+100;
{end sub1}
produce sub2
int x,y;
produce sub3
x=150;
y=x-200;
z=y-x;
Call sub1;
x=y+z;
{end sub3}
begin {sub2}
x=100;
call sub3;
z=y+100;
{end sub2}
main
 x=20;
 y=50;
 z=15;
 call sub2;
 print(x,y,z)
```

Dinamik Kapsam

A	sub1 (y,z)	sub2 (x,y)	sub3
x=20	~~y=0~~ 150	~~x=100~~ ~~150~~ 400	-250
y=50	~~z=0~~ 170	y=-50	
~~z=15~~ ~~-200~~ 50			

1)- sub2'ye dallan x ve y tanımlanmış.
x=100

2) sub3'e dallan hiçbir şey tanımlanmamış
x=150 demiş ama sub2'den geldiğin için onu değiştir.
y=x-200 → y=150-200 y=-50
z=y-x → z=-50-150 = -200
sub2'de z tanımlamadığı için A'yı değiştir.

3) sub1'e dallan y ve z tanımlanmış.
y=0 z=0
y=x+z → y=150+0
y=150
z=y+20 → 150+20=170
x=x+y+100 → sub2'deki x ile
150+150+100 = 400

4) Tekrar sub3'e paldım.
x=y+z
-50-200 = -250

5) Tekrar sub2'ye paldım
z=y+100
-50+100
z=50 → A'daki z 50 olur.

Çıktı → x=20 y=50 z=50

Statik Kapsam

A	sub1 (y,z)	sub2 (x,y)	sub3
~~x=20~~ 140	~~y=0~~ 20	~~x=100~~ 150 10	
y=50	~~z=0~~ 40	y=-50	
~~z=15~~ ~~-200~~ 50			

1) sub2'ye dallan → x=100
2) sub3'e dallan hiçbir şey tanımlanmamış. x=150
sub2'deki x'i değişecektir çünkü üst program
y=x-200 sub2'deki y'ye -50 yaz.
=150-200
=-50

z=y-x z=-50-150 =-200
üst program A A'daki z'ye -200 yaz.

3) sub1'e dallan. (y,z)
y=0 z=0
y=x+z → üst program A değişecek
20+0
y=20
z=y+20
=20+20=40
x=20+20+100
=140

4) Tekrar sub3'e pal.
x=y+z
-50+40
=-10

5) Tekrar sub2'ye pal.
z=y+100
z=-50+100 =50
A'daki değişecek

Çıktı → x=140 y=50 z=50

ÖRNEK

```
program main;
procedure sub1;
var a,y : integer;
begin {sub1}
a = x+z+10;
y = 100;
x = a+20;
end {sub1}
procedure sub2;
var a,x : integer;
procedure sub3;
var a,z : integer;
begin {sub3}
a=y;
x = a+10;
z = 500;
call sub1;
y = x-z;
end {sub3}
begin {sub2}
a=5;
call sub3;
z=z+x+100;
end {sub2}
begin {main}
x=77;
y=77;
z = x+y;
call sub2;
print x,y,z;
end {main}
```

(x,y,z) main	(a,y) sub1	(a,x) sub2	(a,z) sub3
x=77	a = 597	a=5	a=77
~~y=77~~ 117	y=100	~~x=87~~	z=500
~~z=154~~ 871		x=617	

617 - 500 = 117

z = z+x+100 → 154 + 617 + 100 = 871

Çıktı ⇒ x=77 y=117 z=871

Statik Kapsam

(x,y,z) main	(a,y) sub1	(a,x) sub2	(a,z) sub3
~~x=77~~ 261	a=~~77~~+154+10	a=5	a=77
~~y=77~~ -413	a=241	x=87	z=500
~~z=154~~ 341	y=100		

z = z+x+100 = 154+87+100 = 341

y = x - z = 87 - 500 = -413

Çıktı ⇒ x=261 y=-413 z=341

Bölüm 6

* Bir problemin çözümü esnasında bilgisayarda tutulan, CPU komutu olmayan her türlü bilgiye veri denir.
* Bir veritipi, bir değerler kümesini ve bu değerler üzerindeki işlemleri tanımlar.

- ilkel (temel) ve yapısal veri tipleri arasındaki en önemli fark, ilkel veri tiplerinin başka veri tiplerini içermemesidir.

İlkel veri tipleri → integer, float, decimal

İlkel Veri Tipleri

Başka veritipleri aracılığıyla tanımlanmayan veri tiplerine ilkel (primitive) veri tipleri denir.

Yapısal Veri Tipleri

Yapısal tipler ilkel tiplerden oluşur ve bellekte bir dizi yerleşimde saklanırlar.

Yapısal Veri Tipleri → Diziler, Kayıtlar, Göstergeler

Record (Kayıt Tipi)

Kayıt (record) tipi, her elemanın ismiyle ayırt edildiği heterojen veri elemanları topluluğudur.
Dizilerde → homojen elemanlar Kayıtlarda → heterojen elemanlar

Union (Bileşim) Tipi

Bir programın çalışması boyunca farklı değerler saklayabilen bileşim (union) tipi, kayıt tipinin özel bir şekli olarak görülebilir.

set (Küme) Tipi

Bir küme (set) tipi, sıralı bir tipin sıralı olmayan değerlerini saklayabilir. Küme tipleri, sayısal kümeleri modellemek için kullanılır.

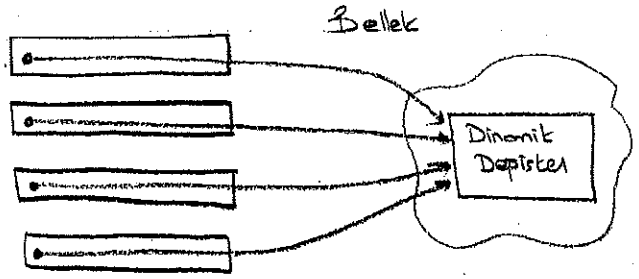
Pointer (Gösterge) Tipi

- Gösterge (pointer) tipi, belirli bir veriyi içermek yerine başka bir veriye başvuru amacıyla kullanılır. Sadece bellek adreslerinden oluşan değerler ve boş değerini içerebilen bir tiptir.

Dangling Pointer (Sallanan gösterge)

Serbest bırakılmış bir bellek adresini gösteren değişkene sallanan gösterge denir. Bir göstericinin gösterdiği belleğin bir şekilde sisteme iade edildiği durumdur.

* Java'da gösterge veritipine dilde yer verilmeyerek, güvenilirlik problemlerinin önlenmesi amaçlanmıştır.
* Pointer, erişilebilecek bellek yerini genişletir.



Kuvvetli tipleme

- Farklı veri tiplerinin farklı sayıltmaları göstermeleri nedeniyle etkileşimlerinin kısıtlanmasıdır.
- Bir programlama dilinin kuvvetli tipli bir dil olarak nitelenmesi için, dilin tüm tip hatalarını yakalanması gerekir.

ÖRTÜLÜ TİP DÖNÜŞÜMLERİ

Aşağıdaki programda a=b*d ifadesindeki d'nin yanlışlıkla yazıldığını düşünelim.

```
void main()
{
int a,b,c;
float d;
a=b*d;
}
```

Derleyici, b yi float tipine dönüştürerek ve çarpım işlemi float olarak gerçekleşecektir. Zorunlu dönüşüm, tip uyuşmazlığı hatasını engellemiştir.

- Güvenilirlik olumsuz etkileniyor.
- Hataların fark edilmesini engelleyebilir
- Zorunlu dönüşümün gerçekleştiği dillerde, tip denetimi kısıtlıdır.

DIŞSAL TİP DÖNÜŞÜMLERİ

Hem daralan hem de genişleyen özellikte dışsal tip dönüşümleri gerçekleşebilir. İşlemden önce parantez içinde değerin dönüştürülmesi istenen tip yazılır

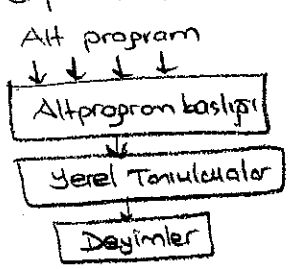
```
#include <stdio.h>
void main()
{
double x;
x = 4/3;
printf("sonuç=. %f\n", x);
}
```

Burada x değişkeni double olarak tanımlanmıştır. Double veri tipi kesirli ve çok büyük sayıları tutma özelliğine sahiptir. x=4/3; işlemi ile 4 sayısı 3'e bölünmekte ve sonuç x değişkenine aktarılmaktadır ve printf komutu ile ekrana sonuç = 1.00000 olarak yazdırılmaktadır. Bunun sebebi 4 ve 3 sabit bilgilerinin int veri tipinde birer bilgi olmasındır

```
#include <stdio.h>
void main()
{
double x;
x = 4.0/3;
printf("sonuç= %f\n", x);
x = (double) 4/3;
printf("sonuç= %f\n", x);
x = (double)(4/3);
printf("sonuç = %f\n", x);
}
```

BÖLÜM-8

Bir programda birden çok kez etkin duruma getirilebilecek bir dizi deyimin bir isim altında gruplanması ile alt programlar oluşturulur.
- Alt programın bir tane giriş yeri olur.
- Alt program çağrıldığında ve yürütülürken çağıran program bekler.
- Çağrılan program bittiğinde kontrol her zaman çağıran programa geri döner.



ÖR
procedure ornek (a:integer, b:real)
altprogram ismi — resmi parametreler

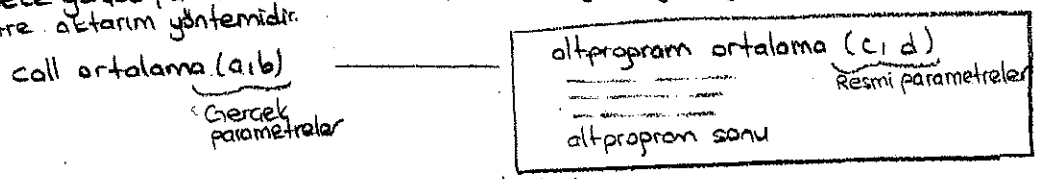
ÖR
call yordam (a,b,c)
Çağrı deyimi — alt program ismi — Gerçek parametreler (Yordamın resmi parametrelerine bağlanır)

PARAMETRE AKTARIM YÖNTEMLERİ

* Resmi parametreler ve gerçek parametreler arasındaki veri akışı, parametre aktarım yöntemlerine göre gerçekleştirilir.

Çağırma Yöntemleri
- Değer ile çağırma
- Sonuç ile çağırma
- Değer ve sonuç ile çağırma
} Gerçek değer fiziksel olarak kopyalanarak aktarılır.
- Başvuru ile çağırma
- İsim ile çağırma
} Verinin erişim yolu aktarılır.

Değer ile Çağırma (call by value)
- Gerçek parametre, sabit, değişken veya ifade olabilir.
- Sadece gerçek parametreden resmi parametreye değer geçişi olduğu için en güvenilir parametre aktarım yöntemidir.

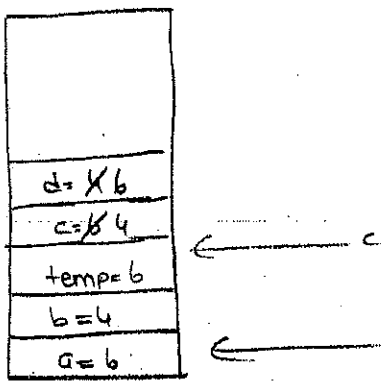


* a ⟶ c (a'nın değeri c'ye aktarılır)
* c yerel bir değişken olarak nitelendirilir.

ÖRNEK

```
main(){
int a = 6;
int b = 4;
Cswap (a,b);
//a=6
//b=4
}

Cswap (int c, int d){
 int temp = c;
 c = d;
 d = temp;
```



* Gerçek parametre, resmi parametrenin yığındaki yerine kopyalanır.
* Resmi parametrenin değişimi gerçek parametreye yansımaz.

Sonuç ile Çağırma

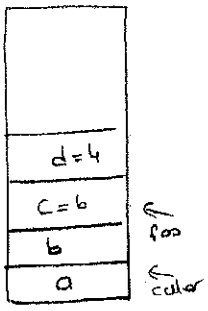
* Gerçek parametrenin değişken olması zorunludur.
* Gerçek parametreye karşı gelen resmi parametre, altprogramın çalışması süresince yerel değişkendir.
* Bu yöntem uygulamasındaki güçlük, gerçek parametrenin değerinin resmi parametreye aktarılmasının önlenmesidir.

call ortalama (a,b) ——→ altprogram ortalama (c,d)
(Gerçek parametre) (Resmi parametre)

* c'nin değeri altprogram sonunda a'ya aktarılır. c ——→ a //

ÖRNEK

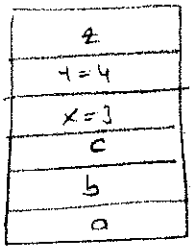
```
caller (){
 int a;
 int b;
foo (a,b);
//a=6
//b=4
   }
foo (int c,int d){
 c=b;
 d=4;  }
```



a=6 b=4

ÖRNEK

```
void plus (by_value int a, by_value int b, by result int c)
{
 c=a+b;
 }
void f ()
 {
int x=3;
int y=4;
 int z ;
plus (x,y,z);
write z;
 }
```



c=3+4
c=7
z → 7

Çıktı: 7 //

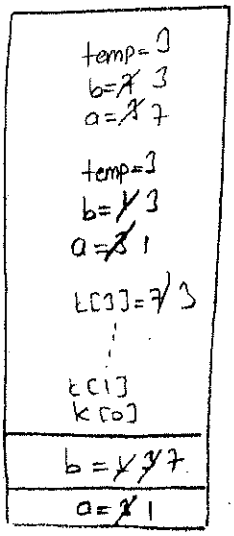
Değer ve Sonuç ile Çağırma

Bu yöntemde, gerçek parametrenin değeri ile karşı gelen resmi parametrenin değeri ilklendikten sonra resmi parametre, alt programın çalışması süresince yerel değişken gibi davranır ve alt program sona erdiğinde resmi parametrenin değeri gerçek parametreye aktarılır. Bu yöntemde de gerçek parametrenin değişken olması zorunludur.

ÖRNEK

```
integer a=3;
integer b=1;
integer k[10];
k[3]=7;
swap (a,b);
swap (b,k[b]);
procedure swap (a: in out integer;
                b: in out integer ) is

 temp : integer;
 begin
 temp := a;
 a := b;
 b := temp;
 end swap;
```



Başvuru ile Çağırma (Call by Reference)

En önemli farkı, altprograma verinin adresinin aktarılmasıdır. Bu adres aracılığıyla altprogramı çağıran program ile aynı bellek yerine erişebilir ve gerçek parametre, çağıran program ve altprogram arasında ortak olarak kullanılır.

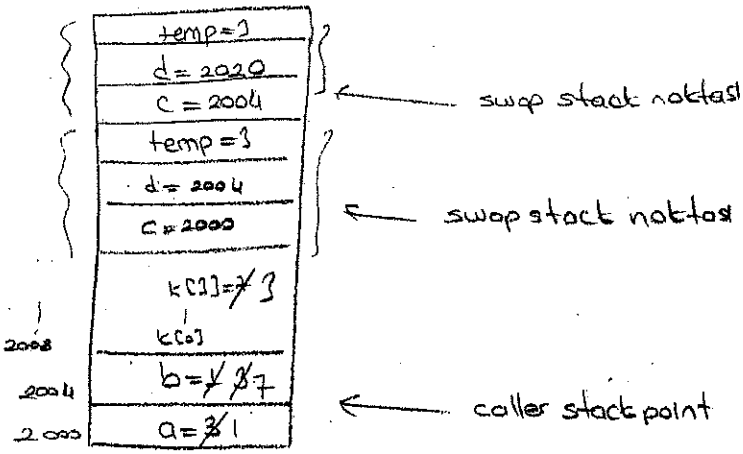
call ortalama (a,b) → altprogram ortalama (c,d)
gerçek fonk. — — — Resul
son

- Alt programa a'nın değeri değil, adresi aktarılır.

* Parametre olarak bellek adresi geçirilir.
* c*, a ile gösterilen bellek bölgesini gösteren bir işaretçidir.

ÖRNEK

```
caller () {
int a = 3;
int b = 1;
int k[10];
k[3] = 7;
swap (&a, &b);
swap (&b, &k[b]);
}
swap (int *c, int *d){
temp = *c;
*c = *d;
*d = temp;
}
```



İsim ile Çağırma (Call by Name)

- Bir gerçek parametre isim ile çağırma yöntemi ile aktarıldığında, altprogramda gerçek parametreye karşı gelen resmi parametrenin bulunduğu her yere metinsel olarak gerçek parametre yerleştirilir.
- Eğer gerçek parametre bir sabit değerse, isim ile çağırma yöntemi, değerle çağırma yöntemi ile aynı şekilde gerçekleşir.
- Eğer gerçek parametre bir değişkense, isim ile çağırma yöntemi, başvuru ile çağırma yöntemi ile aynı şekilde gerçekleşir.

ÖRNEK

```
function sub (int a, int b, int c)
begin
a := b+c;
b := c+1;
print a,b,c;

function main
begin
int i := 5;
int j := 10;
int k := 15;
sub(i, j, j+k);
print (i, j, k);
end
```

Alt programda yaptığımız işlemler, ana programı başlatınız iki tane printf var. Bi' altprogramda bide main'de.

(5, 10, 15) → (i, j, j+k) → (5, 10, 25)

a = b+c → a = 35
b = c+1 → b = 26
c = 25

Çıktı

35	26	25	→ sub
5	10	15	→ main

* a ve b call by (reference), c ise call by value
i'nin adresini a, j'nin adresini b tutuyor.

Çıktı:

35	26	25
35	26	15

k değişkenini tutan bir depo/ster yoktu alt programda

* a ve b call by value result (değer sonuç), c ise call by value (değ)
klasik alt program çağırma mantığı, püf nokta: j+k'yı gönderdiğimiz için arası yine değerle çalışmaya göre çalışacak

Çıktı:

35	26	25
35	26	15

* a, b, c'nin hepsi call by name (isim) ise
ismi gönderiyoruz, her değişiklik anında o ismin tuttuğu adreste belirtiliyor (önemli olan c)

a = 35 b = 26 → b, j'nin değerini tutuyordu.
j+k = 26+15 = 41

Çıktı:

35	26	41
35	26	15

ÖRNEK

```
int i = 1;
void p (int f, int g){
g++;
f = 5*i;
}
int main (){
int a[] = {0,1,2};
p (a[i], i);
printf ("%d %d %d %d\n", i, a[0], a[1], a[2]); }
```

Değer ile çağırma

i	a[0]	a[1]	a[2]	f	g
1	0	1	2	1	1
				5	2

Çıktı: 1 0 1 2

Başvuru ile çağırma

i/g	a[0]	a[1]/f	a[2]	f	g
1	0	1	2		
2		10			

Çıktı: 2 0 10 2

İsim ile çağırma

i/g	a[0]	a[1]/f	a[2]
1	0	1	2
2			10

Çıktı: 2 0 1 10

Çünkü i'yi arttırdık i[2]'ye yazarız
o yüzden.
a[i] ifadesi ihtiyaç duyulmayana kadar
değiştirilmez, bu durumda i değiştirildikten sonraya kadar.

ÖRNEK

```
int i=2;
void foo (int f, int g){
f=f+g;
g=g+4;}
int main(){
int a[]={1,1,1,1};
foo (i, a[i-2]);
printf ("%d %d %d %d %d\n", i, a[0], a[1], a[2], a[3]);
```

İsim ile çağırma

i	a[0]	a[1]	a[2]	a[3]
2	1	1	1	1
3		5		

Çıktı: 3 1 5 1 1

Değer ile çağırma:

if	a[0]	a[1]	a[2]	a[3]
2	1	1	1	1
3	5			

Çıktı: 2 1 1 1 1

Referans ile çağırma:

i/f	a[0]	a[1]	a[2]	a[3]
1	1	1	1	1
3	5			

Çıktı: 3 5 1 1 1

ÖRNEK

```
function sub (int a, int b, int c)
begin
b = a+c;
a = c+1;
print a, b, c;
end
function main
begin
int i = 3;
int j = 5;
int k = 8;
sub (i, j, j+k);
print i, j, k;
end
```

Call by value

i → a (3), j → b (5), j+k → c (13)

14	16	13
3	5	8

b=a+c
b=16
a=c+1
a=14

call by reference

ik'yi tutan bir parametre yok

14	16	13
14	16	8

call by results

14	16	13
14	16	8

call by none

14	16	24
14	16	8

i → a a=3
j → b b=5
j+k → c c=13

b = a+c b=16
a = c+1
a = 14

(j) b=16 olduğu için

j+k → c olduğu için

16+8=24
c=24 oldu

PROGRAMLAMA DİLLERİ (ÇIKMIŞ SORULAR)

1) identifier (<id>) kümesi = {A, B, C} olan sadece toplama ile çarpma işlemlerini içeren atama kurallarından oluşan bir dilin gramerini yazınız.
A = A+B*C atama işlemini yazılan gramer kurallarından türetin.
(NOT: Çarpma işlemi öncelikli)

Çözüm

A = A+B*C

Gramer

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

Sola Dayalı Türetme

<assign> → <id> = <expr>
→ A = <expr>
→ A = <id> + <expr>
→ A = A + <expr>
→ A = A + <id> * <expr>
→ A = A + B * <expr>
→ A = A + B * <id>
→ A = A + B * C //

2) Darboğaz nedir ? Açıklayınız.

Cevap:

Bir bilgisayarın belleği ve işlemcisi arasındaki bağlantı hızı o bilgisayarın hızını belirler. Program komutları çoğu kez bu bağlantı hızından çok daha hızlı yürütülebilir : bağlantı hızı bu yüzden darboğaza sebep olur.

3) Break ve continue komutları arasında ne fark vardır ?

Cevap:

* Break komutu ile switch sonuna gidilir, ordan default çıkışa erişilir direk. Döngü bitirilir.
- Ama continue döngüyü bitirmez, kontrol kısmına gönderir.

4) Eğer 5 bitlik mimaride işaretli sayılar iki'ye tümler yöntemi ile temsil ediliyorsa −15 ile +16 sayılarının toplama işleminin nasıl gerçekleştirildiğini gösteriniz.

Çözüm:

15 → 01111
10000
+
-15 → 10001

16 → 10000
-15 → 10001
+
00001
1 → 00001 //

5) Kendisine parametre olarak gelen 2 tane değişkenin içeriklerini yer değiştiren bir C veya C++ alt programı yazınız. Alt program çalışmasını bitirip, çağıran programa dönüldüğünde yapılan değişiklik geçerli olmalıdır. Hem alt programı hem de nasıl çağrılacağını kodlayınız.

(NOT: C veya C++ default parametre geçirme yöntemi değer ile çağrılmalıdır).

Cevap:

①
```
void swap2 (int *a, int *b) {
int temp = *a;
*a = *b;
*b = temp; }

swap2 (&c, &d) }
```

②
```
ana program () {
int a = 3;
int b = 4;
swap (a,b); }

swap(int c, int d) {
int temp = c;
    c = d;
    d = temp; }
```

} Referans ile çağırma yöntemi

6)

```
k == (J+13)/27
loop :
if k>10 then goto out
k := k+1
i == 3*k-1
goto loop
out : ..
```

Cevap:

```
for (k= (J+13)/27  k<=10  k=k+1) {
    i = 3* k-1
}
```

Program parçacığına karşılık gelen döngü yapısını C/C++ veya Java kodunda yazınız.

```
for (k=(J+13)/27 ; k>10 ; k = k+1) {
    i = 3*k-1;
}
```

7) C dilinde aşağıdaki fun isimli bir fonksiyon ve bu fonksiyonun kullanıldığı main programı tanımlanmıştır. a ve b şıklarındaki öncelik durumlarında sum1 ve sum2 değişkenlerinin değerlerini hesaplayınız.

```
int fun(int *k){
    *k += 4;
    return 3*(*k)-1;
}
void main(){
int i=10, j=10, sum1, sum2;
sum1 = (i/2) + fun(i);
sum2 = fun(&j) + (j/2);
}
```

Cevap

a) Soldan sağa doğru

sum1 = (10/2) + 41 = **46**

sum2 = 48

b) Fonksiyon referansı öncelikli

sum1 = 48

sum2 = 48

8) Aşağıdaki pascal program parçacığına karşılık gelen kodu if yapısı kullanarak C, C++ veya Java dillerinden birisinde yazınız.

```
case index-1 o;
2,4 : even := even+1;
1,3 : odd := odd+1;
0: zero := zero+1;
else error:= true
end
```

Cevap

```
input = index-1;
if (input==2 || input==4)
    even += 1;
else if (input==1 || input==3)
    odd += 1;
else if (input==0)
    zero += 1;
else
    error = true;
```

9) C dilinde aşağıdaki gibi bir program parçacığı yazılmıştır. Kod çalıştıktan sonra j değişkeninin alacağı değeri hesaplayınız.

```
j=-3;
for(i=0; i<3; i++){
  switch(j+2){
      case 3:
      case 2: j--; break;
      case 0: j+=2; break;
      default: j=0;
  }
  if (j>0) break;
     j=3-i;
}
```

Cevap

i	j
0	0
0	3
1	0
1	2
2	0
2	1

j = 1

4

10) İmperatif dillerin temel özellikleri nelerdir?

Cevap:

- Depisteler
- Atama İfadeleri
- İterasyon

11) Bir programlama dilini değerlendirirken hangi kriterlerden bahsetmek gerekir? C ve Java dillerini bu kriterlere göre kıyaslayınız.

Cevap:

* Okunabilirlik (Readability): Programın okunabilme ve anlaşabilme kolaylığı.
* Yazılabilirlik (Writability): Bir dilin program yazmada kullanılabilme sözdizim"
* Güvenilirlik (Reliability): Şartlara uygunluk, şartları sağlama.
* Maliyet (Cost): En son toplam maliyet.
* Ortogonallik: Elimizdeki yapılarla kombinasyonlarla birleştirip yeni yapılar oluşturulabilir

	Java	C
Okunabilirlik	+	+
Yazılabilirlik	+	+
Güvenilirlik	+	—
Maliyet	+	—
Ortogonallik	+	+

12) Derleyici ile yorumlayıcı arasındaki farklar?

Cevap:

Yorumlayıcı

- Tüm programı deyim deyim denetler.
- Bir döngü içindeki tüm deyimler her defasında çevrilir.

Derleyici

- Tüm programı bir kerede denetler.
- Sözdizimsel hataları bulur.
- Hata yok ise, programı nesne koda çevirir.
- Nesne kod daha sonra çalışabilir koda çevrilir.

13) Betimleyici ne demektir? Açıklayın.

cevap:

- Derleyici (compiler) anında dizinin başını ve sonunu hafızada belirtmesi için tutan yapıdır.

14) Terimlerin açıklamalarını yapınız.

Scope ⇒ (Komutların içindeki alandır)
- Bir değişkenin kapsamı değişkenin yürütülebilir olduğu komutların içindeki alandır.

Lifetime ⇒ (Koparılma süresi)
- Bir değişkenin belirli bir bellek hücresiyle bağlanması ile bu bağın koparılması arasında geçen süredir.

Type Checking ⇒ (Tip hatalarının testi)
- Programın çalışması veya derlenmesi süresince verilen programın tip hatalarının test edilmesi işlemidir.

Binding ⇒ (Eleman arası ilişki (bağlama))
- Bir özellikle, bir program elemanı arasında ilişki kurulması (binding) bağlama denir. Bağlama zamanı statik ve dinamik olması ile çeşitlilik gösterir.

15) Bellek sızıntısı (Memory leak) ve Sallanan Pointer (Dangling) (Göstergeç) kavramlarını bir örnek üzerinde açıkla.

Cevap:

Dangling Pointer (Sallanan Göstergeç)
- Bir göstergeçte değişkenin gösterdiği adreste geçerli veri olmaması durumunda, göstergeçin serbest bırakılmış bir dinamik yığın değişkene işaret etmesi ile oluşur, serbest bırakılmış bellek adresini gösteren değişkene dangling pointer denir.

Ör

```
int n = 11;
int *a, *r, *h;
a = &n;   r = NULL;
h = (int*) malloc(sizeof(int));
*h = 33;
r = h
free(h);
```

r = boştu
h = 33 iken 33 → r'ye geçti,
bu durumda h boşta kaldı.

Bellek Sızıntısı (Memory Leak)
- Bir programdaki dinamik bellek kullanım mantığındaki hataya istinaden oluşan bir bellek erişim arızası. Belleğin kullanımı bitmesine rağmen, belleğin serbest bırakılmamasına sebebiyet verir. Sonda bellek yetersizliği sebebiyle program sonlanır.

```
int n = 11;
int *a, *r, *h;
a = &n;   r = NULL;
h = (int*) malloc(sizeof(int));
*h = 33;
h = NULL;
```

r boşta idi, h boş oldu ama 33 öbek gitti; bellek sızıntısı.

Çözüm için yöntemler
1) Yazıt = Ek bellek hücresidir.
2) Kilit ve Anahtar = Gösterici çiftinden oluşur.

16) Statik ve dinamik tip bağlamasını birer cümle ile açıklayıp dil örneği veriniz?

Cevap:

Statik Tip Bağlama: Tiplerin isimlerle derleme zamanında bağlandığı diller olarak nitelendirilirler. Burada derleyici, tip hatalarını, program çalıştırılmadan önce yakalar.

Dinamik Tip Bağlama: Bir programlama dilinde bir değişkenin tipi çalışma zamanında, değişkenin bağlandığı değer ile belirlenir.

17) Fonksiyonel yan etki kavramını bir örnek üzerinde gösteriniz.

Cevap:

Bir fonksiyon iki yönlü bir parametreyi veya lokal olmayan bir değişkeni değiştirdiğinde meydana gelir.

- Bir ifadede çağrılmış bir fonksiyon ifadenin başka bir operantını değiştirdiğinde ortaya çıkar;

```
int fun (int *u) {
    *u = *u/2;
    return *u; }

a=10;
b=a+fun(&a);
//fun, parametresini
değiştiriyor.
```

2 muhtemel çözüm

1) Dil tanımlaması yapılır. İki yönlü parametre ve global değişken olmayacak.

2) Operantların işlem sırasını belirlemek için dil tanımlaması yapılır.

18) Aşağıdaki gramerin belirsiz olup olmadığını ispatla.

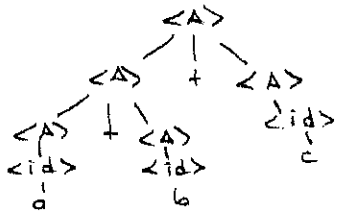
Cevap

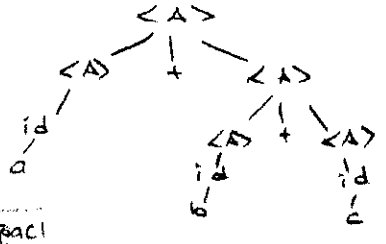
<S> → <A>

<A> → <A>+<A> | id

<id> → a | b | c

a+b+c işlemini yapalım.





İki türlü türetme ağacı olduğu için belirsizdir.

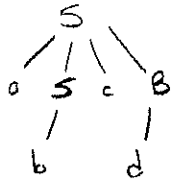
19) Aşağıdaki gramer ile abcd ve acccbd ifadelerinin türetilip türetilemeyeceğini parse ağacı çizerek gösteriniz.

Cevap

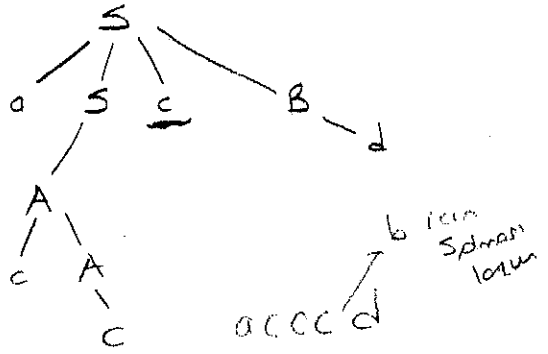
S → aScB | A | b

A → cA | c

B → d | A



* abcd oluşur.



* acccbd oluşmaz.

20)
```
program A() {
  x,y,z : integer;
  procedure B()
  {
  y : integer;
  y = 0;
  x = z+1;
  z = y+2; }
  procedure C()
  {
    z : integer;
      procedure D()
      {
        x : integer;
        x = z+1;
        y = x+1;
        call B();
      }
     z = 5;
     call D();
          }
     x = 10;
     y = 11;
     z = 12;
     call C();
     print x,y,z; }
```

STATIK KAPSAM

(x,y,z) A	(y) B	(z) C	(x) D
~~x=10~~ 13	y=0	z=5	x=6
~~y=11~~ 7			
~~z=12~~ 2			

X = 13 y = 7 z = 2

DINAMIK KAPSAM

(x,y,z) A	(y) B	(z) C	(x) D
x=10	y=0	~~z=8~~ 2	x= 6
~~y=11~~ 7			
z=12			

X = 10 y = 7 z = 12

21)
```
function sub (int a, int b, int c)
begin
    b := a+c ;
    a := c+1 ;
    print a,b,c ;
end
function main
begin
    int i := 3 ;
    int j := 5;
    int k := 8;
    sub (i, j, j+k);
    print i,j,k ;
end
```

a	b	c
1	1	1
3	3	3+k
3	5	13

Call by value

14	16	13
3	5	8

* k'yı tutan bir adres yok (değişken değil)

Call by result

14	16	13
14	16	8

Call by reference

14	16	13
14	16	13

Call by name

14	16	24
14	16	8

22) a) 3 nonterminal, 4 terminal ve 5 kurala sahip bir CFG gramerı açıklayacak yaz.

$A \to aB \mid cd$
$B \to a \mid d$
$C \to b$

A, B, C } Non-terminaller
a, b, c, d } terminaller

b) Lexical analiz ve Syntax analiz kavramlarını birer örnek üzerinden açıklayınız.

Syntax Analiz: Bir bilgisayar programının yapısını irdeleyen ya da belirli bir programlama dilinin sözdizimine uyup uymadığını sınayan analiz.

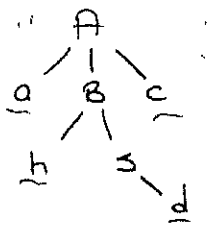
Lexical Analiz (metinsel): Bir derleyicinin ön ucunda yer alan metinsel çözümleyici, bir kaynak programı bir dizi token'a çevirir.

c) Gramerlerde belirsizlik ne demektir. Belirsiz olmayan gramer için örnek bir gramer yazın.

- Bizden istenilen sonucu belirli bir gramere göre iki şekilde veya daha fazla şekilde türetebiliyorsak, bu gramer belirsizdir.

Belirsiz olmayan gramer

$A \to aBc$
$B \to hS$
$S \to d$



ağacı oluştur.

23) BNF kural kümesi:

Kural 1 ⇒ <ifade> ::= <ifade> or <terim> | <terim>
Kural 2 ⇒ <terim> ::= <terim> and <faktor> | <faktor>
Kural 3 ⇒ <faktor> ::= not <faktor> | (<ifade>) | true | false

Programlayıcı kod içerisine "not (true or false)" ifadesi yazmış ise bu ifade için parse tree veya kural kümesinden türetme yaparak düzenli bir ifade olup olmadığını analiz et.

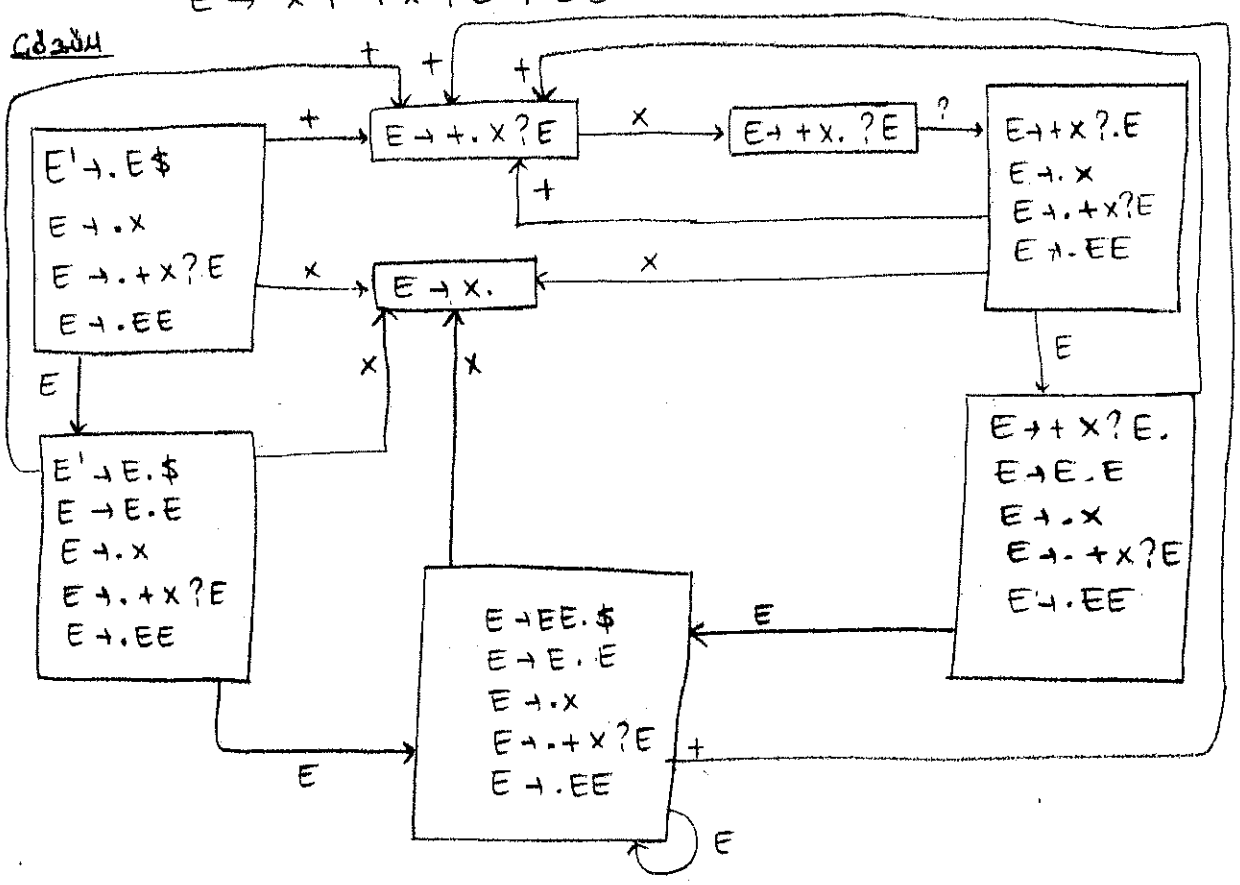
çözüm

= <faktor>
= not <faktor>
= not (<ifade>)
= not (<ifade> or <terim>)
= not (<terim> or <terim>)
= not (<factor> or <term>)
= not (<factor> or <factor>)
= not (true or <factor>)
= not (true or false)

24) Aşağıdaki CFG için LR(0) parsing DFA diyagramını açıklayarak çiziniz.

$E \rightarrow x \mid +x\,?E \mid EE$

Çözüm



25) Aşağıdaki CFG için First ve Follow kümelerini açıklayarak elde ediniz.

$S \rightarrow ABD$

$A \rightarrow a \mid BSB$

$B \rightarrow b \mid D$

$D \rightarrow d \mid \varepsilon$

Çözüm

	First	Follow
S	a, b, d, ε	$, b, d
A	a, b, d, ε	b, d
B	b, d, ε	d, $, b, a
D	d, ε	$, b, d

A → a|λ
B → b|λ
C → c
D → d|λ
E → e|λ

S	a, b, c	$
A	a, λ	b, c
B	b, λ	c
C	c	d, e, $
D	d, λ	e, $
E	e, λ	$

→ İlk durumda A'dan sonra B gelip. B'nin first u, A'nın follow u oluyo. B yerine λ koyabildiğimiz için A'dan sonra C'de gelebilir O yüzden C'nin de first alırız

→ D'ninde E'nin de first kümesinde λ olduğu için. geldik S nin followuna baktık onu yazdık

S → Bb | Cd
B → aB | λ
C → cC | λ

	First	Follow
S	a, c, b, d	$
B	a, λ	b
C	c, λ	d

S'nin first kümesini bulurken;
S → Bb | Cd
B'ye bakttık B'ye gelip a'yı yazdık daha sonra ilk first kümesine λ yazılmayacağı için B'nin yerine de λ yazılabiliyor diye bide b yazdık. Diğerini de (c) aynı şekilde yaptık.

E → TE'
E' → +TE' | λ
T → FT'
T' → *FT' | λ
F → id | (E)

	First	Follow
E	id, (	$,)
E'	+, λ	$,)
T	id, (	+, $,)
T'	*, λ	+, $,)
F	id, (	*, +, $,)

S → [SX] | a
X → ε | +SY | -b
Y → ε | -SXc

	First	Follow
S	[, a	$, +, b, -,], c
X	ε, +, b, -	], c
Y	ε, -	], c, b

"(x+x)*x"
e → e+t | t
t → t*f | f
f → (e) | x

Stack	Giriş	Hareket
$	(x+x)*x$	shift
$(	x+x)*x$	shift
$(x	+x)*x$	reduce f→x
$(f	+x)*x$	reduce t→f
$(t	+x)*x$	reduce e→t
$(e	x)*x$	shift
$(e+	)*x$	shift
$(e+x	)*x$	reduce f→x
$(e+f	)*x$	reduce t→f
$(e+t	)*x$	reduce e→e+t
$(e	*x$	shift
$(e)	x$	reduce f→(e)
$f	$	reduce t→f
$t	$	shift
$t*	$	shift
$t*x	$	reduce f→x
$t*f	$	reduce t→t*f
$t	$	reduce e→t
$e	$	accept

Pairwisedisjoint set test uygula.

$S \to aSb \mid bAA$
$A \to b\ \{aB\} \mid a$
$B \to aB \mid a$

Çözüm:
First S - {a} / First S - {b} $> \phi$ disjoint.
First A - {b} / First A - {a} $> \phi$ disjoint değil. Çünkü b gelmeyip a peld. yineleme yapılabilir ve First {a} da peldabilir.

$b \mid bA \mid aBA \mid a$

First - B {a} / First - B {a} $> \neq \phi$
disjoint değil.
İkisi aynı olduğu için disjoint değil

$S \to ACB \mid Cbb \mid Ba$
$A \to da \mid BC$
$B \to g \mid \lambda$
$C \to h \mid \lambda$

	First	Follow
S	d, g, h, b, a	$
A	d, g, h, λ	h, g, $
B	g, λ	$, g, h, g
C	h, λ	g, $, b, h

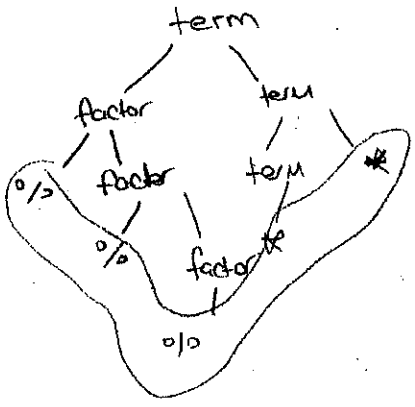
First ve Follow kümeleri verilmiş. Gramer kurallarını bul.

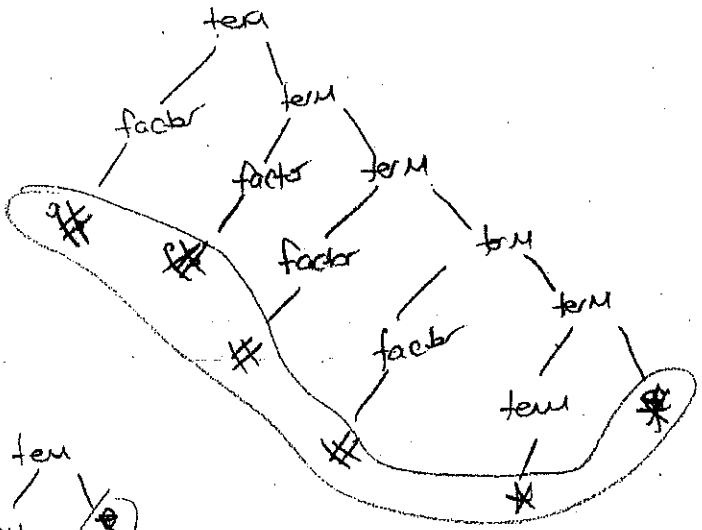
Sembol	First	Follow
S	{0, ⊥}	{0, $}
A	{λ}	{0, ⊥}
B	{λ}	{0, ⊥}

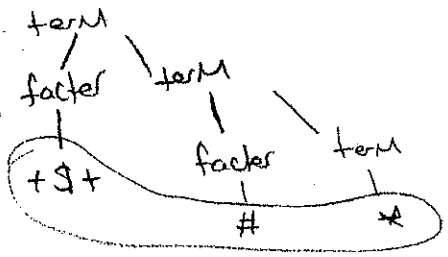
$S \to 0 \mid \bot \mid SO$
$A \to \lambda \mid AS$
$B \to \lambda \mid BS$

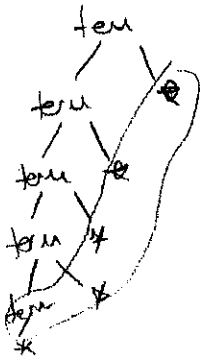
<term> → '*' | '(' <term> ')' '*' | [<factor>] <term>
<factor> → { '+$+' | '#' } | '%' <factor>

% % % * *









SINAV SORUSU

Perl dilinde değişkenin önündeki "my" veya "local" keywordleri, o değişkenin statik (my) ve dinamik (local) scope özelliğinde olduğunu uygulamaktadır. Bu bilgi doğrultusunda aşağıdaki Perl programının ekran çıktısını yandaki boşluğa yazınız.

```
$x = 4;
$y = 5;
print $x, $y "\n";          ------> ilk print'imiz      4 5
&ana;                       ------> ana metoduna git
print $x, $y "\n";          ------> En son dördüncü print   5 5
sub ana {
 local $x = 10;
    my $y=20;
   print $x, $y "\n";       ------> ikinci print        10 20
    &alt;                   ------> alt metoduna git
  - print $x, $y "\n";
 }
sub alt {
    my $x;                  ------> 4 (statik olduğu için kapsadığı kısmın baktım)
    local $y;               ------> 20 (dinamik olduğu için ve alt programına, ana programdan gelliğ için ana programdaki y değerini aldım)
    $x++;                   ------> 5
    $y++;                   ------> 21
}
```

(Tekrar kaldığın yere dön

print diyor (3.) ------> ama ana metoduna tanımladığı için bi daha 10 20

5 21

(Tekrar kaldığın yere dön)

Çıktı

```
4  5
10 20
10 20
4  5
```

SINAV SORUSU

Aşağıdaki program parçalarının optimize edilmiş hallerini yandaki boşluklara yazınız.

```
for (i=0; i<strlen(s); i++){
    s[i]+=5;
}
```

strlen değişken gibi bir şey tanımladığı için bu kod optimize edilemez.

Başka cevap →
```
int uzunluk = strlen(s);
int i;
for (i=0; i<uzunluk; i++)
    s[i]+=5; }
```

```
for (i=0; i<100; i++){
    fonksiyon1(i);
}
```

```
int i;
int a=100;
for (i=0; i<a; i++){
    fonksiyon1(i);
```

```
int kup_alma(int a){
    return a*a*a;
}
```

```
int kup_alma(int a){
    x=a*a;
    a=x*a;
    return a;
```

Başka cevap →
```
int kup_alma(int a){
    int kup=a*a*a;
    return kup;
```

```
int global;
void f(){
int i;          } Kodu kısaltma
i=1;
global=1;       i=1 tanımladığı için global=1 hiçbir işlem arada yapılmayacağından
global=2;
return;
global=3;       Kabul olmaz işlem çünkü en son return diyor
}
```

```
int global;
void f(){
int i=1;
global=2;
}
```

Başka cevap →
```
int global;
void f(){
global=2;
return;
}
```

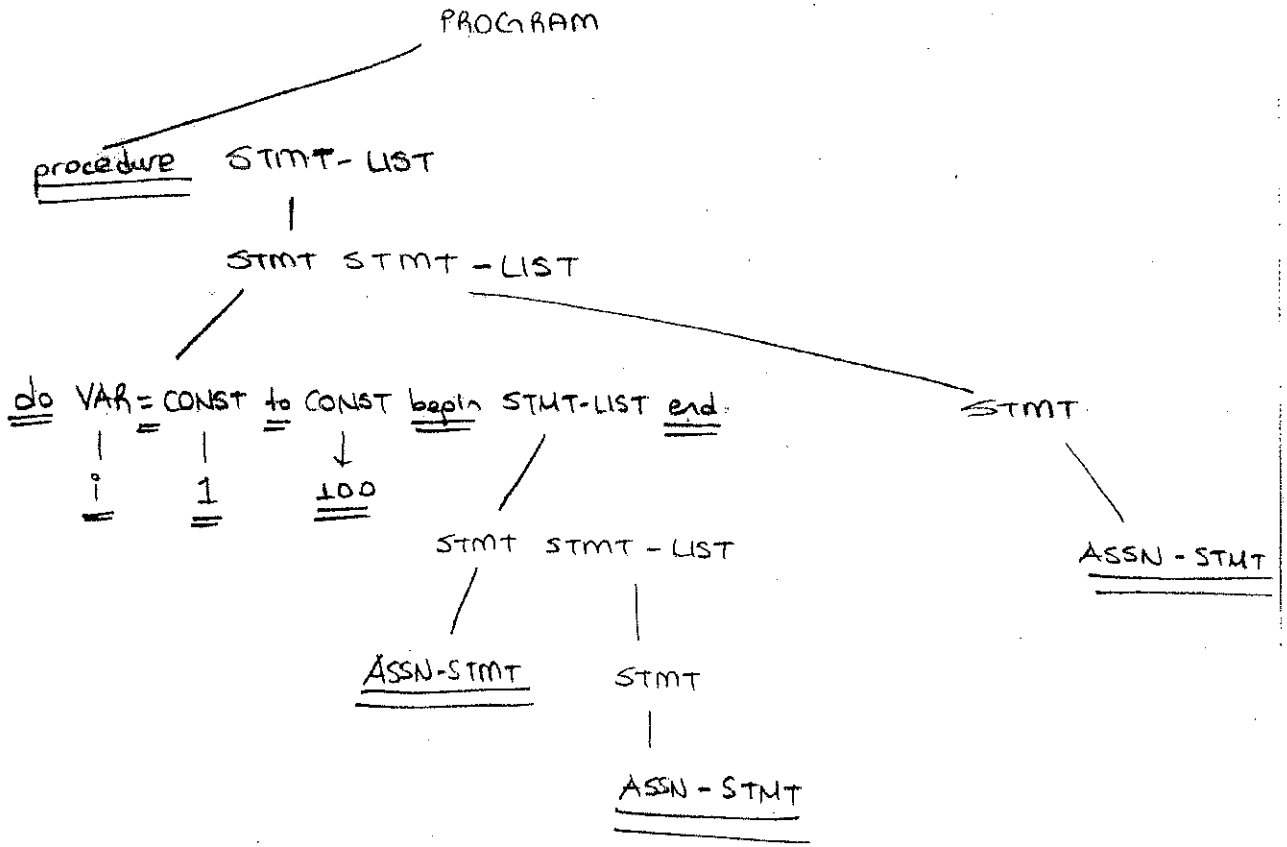
Aşağıdaki context free gramer pascal benzeri bir dilin sentaksını göstermektedir. Terminal olmayan semboller büyük harf ve terminal semboller ise küçük harfler ile gösterilmiştir. VAR değişken ismini ve CONST sabiti ifade etmektedir. Buna göre aşağıda verilen program parçasını üreten ayrıştırma ağacını çiziniz?

PROGRAM → procedure STMT-LIST

STMT-LIST → STMT STMT-LIST | STMT

STMT → do VAR = CONST to CONST begin STMT-LIST end | ASSN-STMT

```
procedure
do i = 1 to 100 begin
      ASSN-STMT
      ASSN-STMT
   end
ASSN-STMT
```



FİRST – FOLLOW SINAV SORUSU

0 ve 1 üzerinde tanımlı bir Context free Gramer S, A ve B gibi üç terminal olmayan sembole sahiptir. Bu gramerin First ve Follow kümeleri aşağıda verilmiştir. Sunuya göre bu gramerin kurallarını veriniz?

Sembol	First	Follow
S	{0,1}	{0,1,$}
A	{λ}	{0,1}
B	{λ}	{0,1}

Cevap

S → 0 | 1 | AS | BS |

A → λ

B → λ

S → 0 | 1 | 1 S 0

A → λ | AS

B → λ | BS

	First	Follow
S	0, 1	$, 0
A	λ	0, 1
B	λ	0, 1

saplıyor

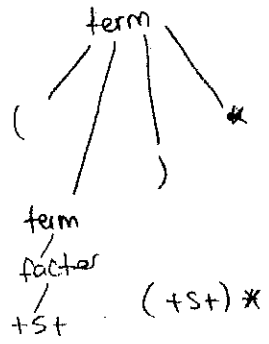
<term> → [illegible]

<factor> → {'+s+' | '#'} | '%' <factor>

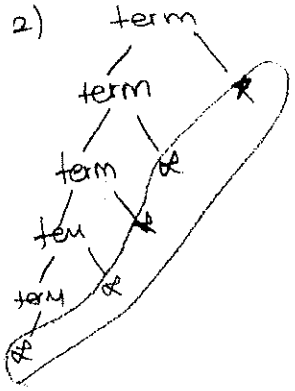
input string	Legal or Not Legal
*	
1) (+$+*)	Legal
2) * ****	Not legal
3) % % % * *	Legal
4) * * * * * **	Legal
5) +$+ * *	Legal

Cây

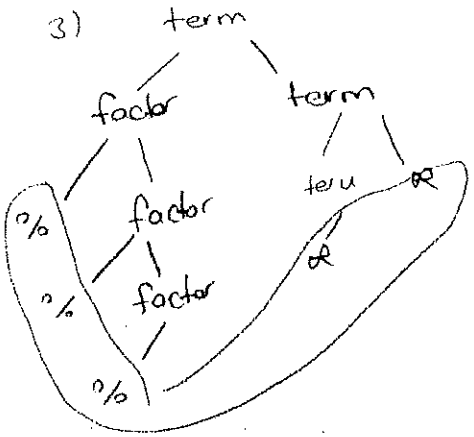
1)



2)



3)



4)