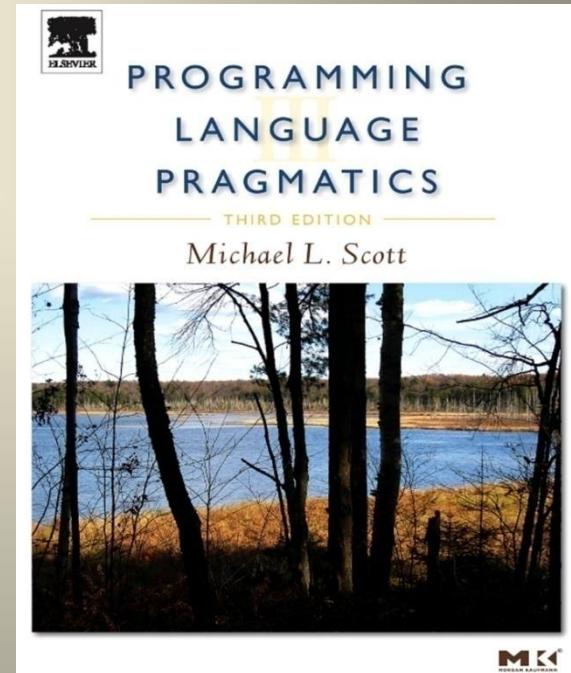
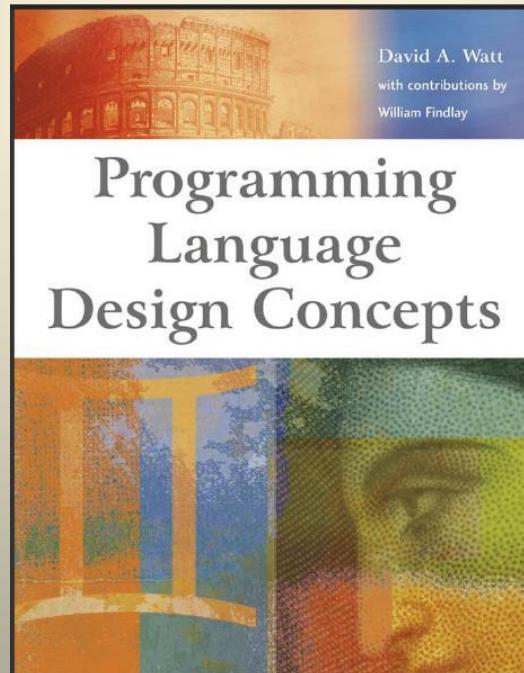
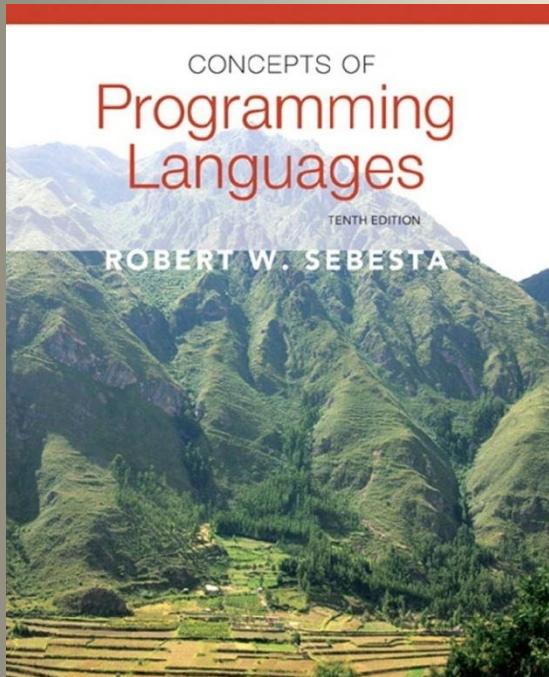


Bölüm: 0

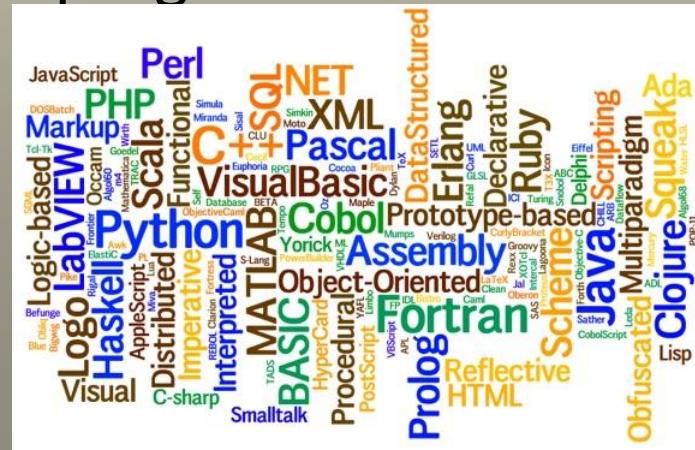


Hazırlık

1. Giriş ve Amaçlar
2. Programlama Dili Nedir?
3. Programlama Dilleri Düzeyleri
4. Dil Çevrimi
5. Programlama Dillerinin Yazılım Yaşam
Döngüsündeki Yeri

0.1. Giriş ve Amaçlar

- Church-Turing hipotezi bütün programlama dillerinin ve bilinen berimsel (hesaplama) cihazların (performans olarak değil, ifade ve problem çözme yeteneği olarak) eşlenik olduğunu söyler.
 - Yani herhangi bir genel amaçlı programlama diliyle yazdığımız bir algoritmayı, başka bir programlama diliyle de yazabiliriz.
 - Peki neden bu kadar çok farklı programlama dili var?
 - Çok fazla sayıda programlama dili bilmenin bir yararı var mı?



Amaç

- Dil öğretmek değil. Bu derste, programlama dillerinin temel kavramları ve bu kavramlar için çeşitli dillerde sağlanan yapılar incelenecaktır. Tanıtılan kavramları örneklemek için, Pascal, C, C++, Java, Lisp, Prolog gibi çeşitli programlama dilleri kullanılacaktır.
- Programlama dillerinin ortak kavramlarını incelemek.
- Farklı paradigmaları, yaklaşımları incelemek.
- Programlama dilinin “daha iyi” olması ne demektir, ne programlama dilini daha nitelikli yapar?
- Derleyici tasarıımı, yazılım mühendisliği, nesneye yönelik tasarım, bilgisayar insan etkileşimi gibi diğer konulara temel oluşturmak.
- Kısaca bu dersin amacı, öğrencinin hem programlama dillerinin temelindeki kavramları öğrenmesini, hem de programlama dilleri alanında kapsamlı bir bakış açısına sahip olmasını sağlamaktır.

Amaç

- Kısaca bu dersin temel amacı; çağdaş programlama dillerinin temel yapılarını tanıtmak ve mevcut, gelecek programlama dillerini değerlendirmek için gerekli araç ve bilgiyi kazanmaktır.
- Burada kazanılan bilgi, derleyici tasarımda, programlama dillerinin sözdizim (syntax) kurallarının tasarlanması ve sözcüksel (lexical) analizinde kullanılabilir.

- Şu ana kadar 2500'den fazla programlama dili yapılmıştır.
(Wikipedia)
 - http://en.wikipedia.org/wiki/Alphabetical_list_of_programming_languages
 - <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>
- «Hello world» kelimesinin farklı dillerde yazılımına ait örnek.
 - http://en.wikipedia.org/wiki/Hello_world_program_examples

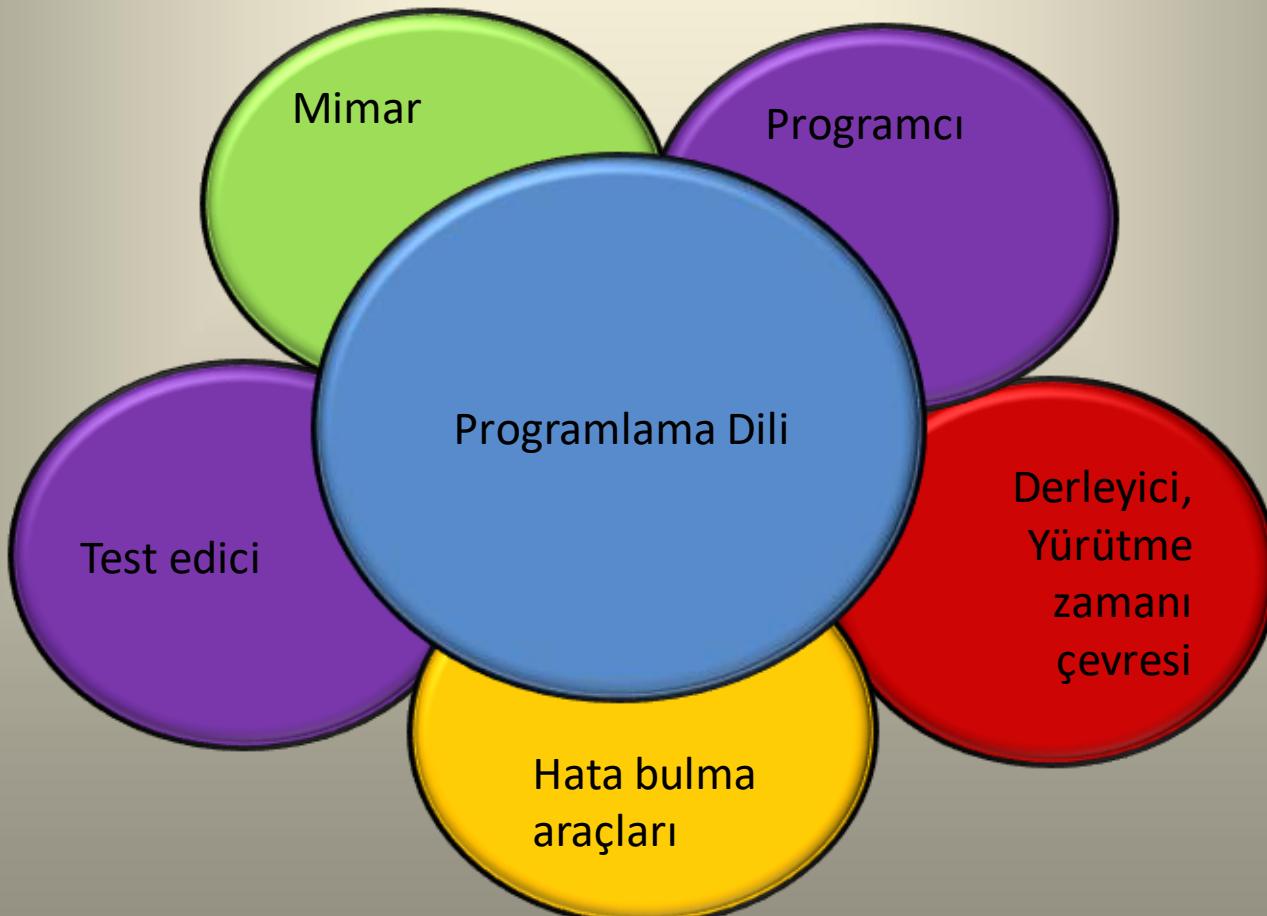
Eylül 2014 için TIOBE Programlama Kominite İndeksi

Eylül	Sep 2013	Change	Programlama Dili	Oranlar	Değişim
1	1		C	16.721%	-0.25%
2	2		Java	14.140%	-2.01%
3	4	▲	Objective-C	9.935%	+1.37%
4	3	▼	C++	4.674%	-3.99%
5	6	▲	C#	4.352%	-1.21%
6	7	▲	Basic	3.547%	-1.29%
7	5	▼	PHP	3.121%	-3.31%
8	8		Python	2.782%	-0.39%
9	9		JavaScript	2.448%	+0.43%
10	10		Transact-SQL	1.675%	-0.32%
11	11		Visual Basic .NET	1.532%	-0.31%
12	12		Perl	1.369%	-0.32%
13	13		Ruby	1.281%	-0.10%
14	-	▲	Visual Basic	1.272%	+1.27%
15	14	▼	Delphi/Object Pascal	1.157%	+0.26%
16	26	▲	F#	0.990%	+0.49%
17	15	▼	Pascal	0.893%	+0.01%
18	-	▲	Swift	0.852%	+0.85%
19	19		MATLAB	0.818%	+0.18%
20	17	▼	PL/SQL	0.809%	+0.13%

Neden Programlama Dilleri Dersi?

- Fikirlerimizi uygularken daha kolay ve daha iyi yapabilmek kısaca etkin algoritmalar geliştirebilmek.
- Seçeneklerimizin ne olduğunu bilirsek iyi seçebiliriz. Programlama dilleri arasındaki seçim kolaylaştırılması için gereklidir.
- Dil öğrenmede yetkinlik. Dillerin özelliklerini bilmeyen, belli bir dille çalışmaya alışmış kişi, farklı bir dili öğrenmesi gerektiğinde zorlanır.
- Belli bir dilin önemli özelliklerini anlayarak daha iyi kullanabilmek , var olan programlama dillerinin kullanımının geliştirilmesi için
- Berimin (Hesaplamanın) gelişmesi için. Dilleri daha iyi değerlendirebilirsek, doğru seçimler yaparız, doğru teknolojilerin gelişmesine destek olmuş oluruz.
- Hata ayıklarken özelliklerini bilmemiz faydalıdır.
- Programlama yapıları hakkındaki bilginimiz artar, özellikleri öğreniriz, olmayan özelliklerine öykünürüz (emulate).
- Yeni bir dil öğrenilmesinin kolaylaştırılması
- Yeni bir dil tasarlamanın kolaylaştırılması

Dil amaçları ve trade-off'lar



0.2. Programlama Dili Nedir?

Doğal Dillerin Genel Yapısı

- Tüm doğal dillerin (Türkçe, Almanca, İngilizce, ... gibi) kendilerine özgü
 - Kelimeleri,
 - İşaretleri,
 - Kelimelerin bir araya getirilip cümlelerin oluşturulmasını sağlayan dilbilgisi kurallarıvardır.
- Dilbilgisi açısından doğru bir cümle oluşturulduğunda, bir anlam ifade etmekte ve herhangi bir kişi o cümleyi anlayabilmektedir.
- Dilbilgisi kurallarında bir hata yapıldığında, oluşturulan cümle anlaşılılamamaktadır.

Programlama Dili Nedir?

- Her dilin, nesneleri, özellikleri, ilişkileri ve işlemleri göstermek için çeşitli semboller ve bu sembollerin birleştirmek için kuralları vardır.
- Bir doğal dil, bir grup insan arasında ortak olarak anlaşılan sembolik bir iletişim dili olarak tanımlanabilir ve iletişim insanlar arasında gerçekleşir.
- Programlama dili ise, insanlar ve bilgisayarlar arasındaki iletişimini sağlarlar.
- Bir **programlama dili**, bir problemin çözümünün bilgisayardaki gerçekleştirimini ifade etmek amacıyla programlar oluşturulması için kullanılan bir dildir.

Doğal Dil - Programlama Dili (1)

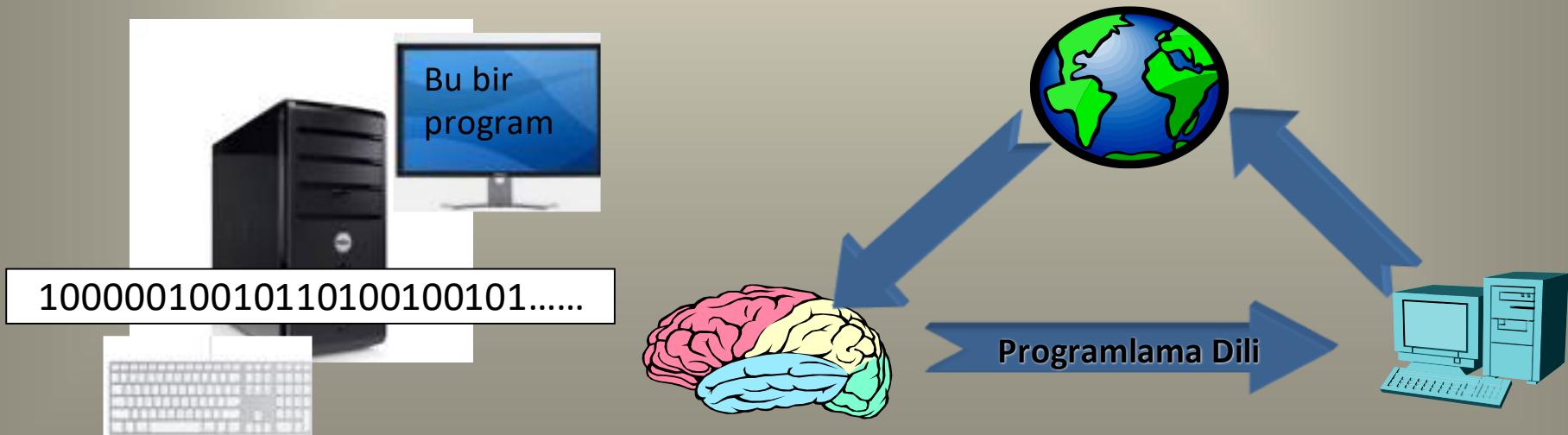
- Bir programlama dili birçok yönden doğal dillere benzer.
- Her programlama dilinin kendisine özgü kelimeleri (söz varlığı, sözlüğü), işaretleri (yazım işaretleri) ve yazım kuralları bulunmaktadır.
- Programlama dillerinde, o dilin dilbilgisi kurallarına o dilin sözdizimi (*syntax*) adı verilmektedir.
- Bir programcı, o anda kullandığı programlama dilinin sözdizim kurallarına göre kelime ve işaretleri biraraya getirerek, bilgisayara birşeyleri yapmasını anlatan, bilgisayar açısından bir anlamaya sahip olan bir deyim oluşturmaktadır.

Doğal Dil - Programlama Dili (2)

- Doğal dillerde konuşma veya yazma sırasında hata yapılabilir.
- Programlama dillerinde hataya yer yoktur.
- Herhangi bir hata yapılması durumunda o deyimin bilgisayar açısından hiçbir anlamı olmamakta, bilgisayar o deyimi anlamamaktadır.
- Bir program yazmak için programcının o dilin tüm sözdizim kurallarını bilmesi gerekmektedir.
- Programcının programlama dilinin kurallarına uymaması durumunda program hatalara sahip olacaktır.
- Bilgisayar hatalı programı anlamayacak, dolayısıyla çalışırmayacaktır.

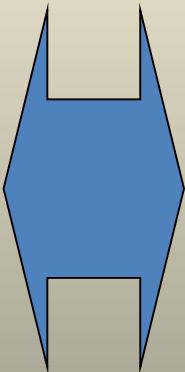
Programlama Dili Nedir?

- **Programlama dili**, bir problemin çözümünün bilgisayardaki gerçekleştirimini ifade etmek amacıyla tasarlanmış ve o programlama dili için **hem insanlar hem de bilgisayarlar tarafından ortak olarak anlaşılacak** kurallar ve semboller dizisidir.



Programlama

```
int sum(int[] x) {  
    int sum = 0;  
    n = 0;  
    while (n < x.length) {  
        sum += x[n];  
    }  
    return sum;  
}
```

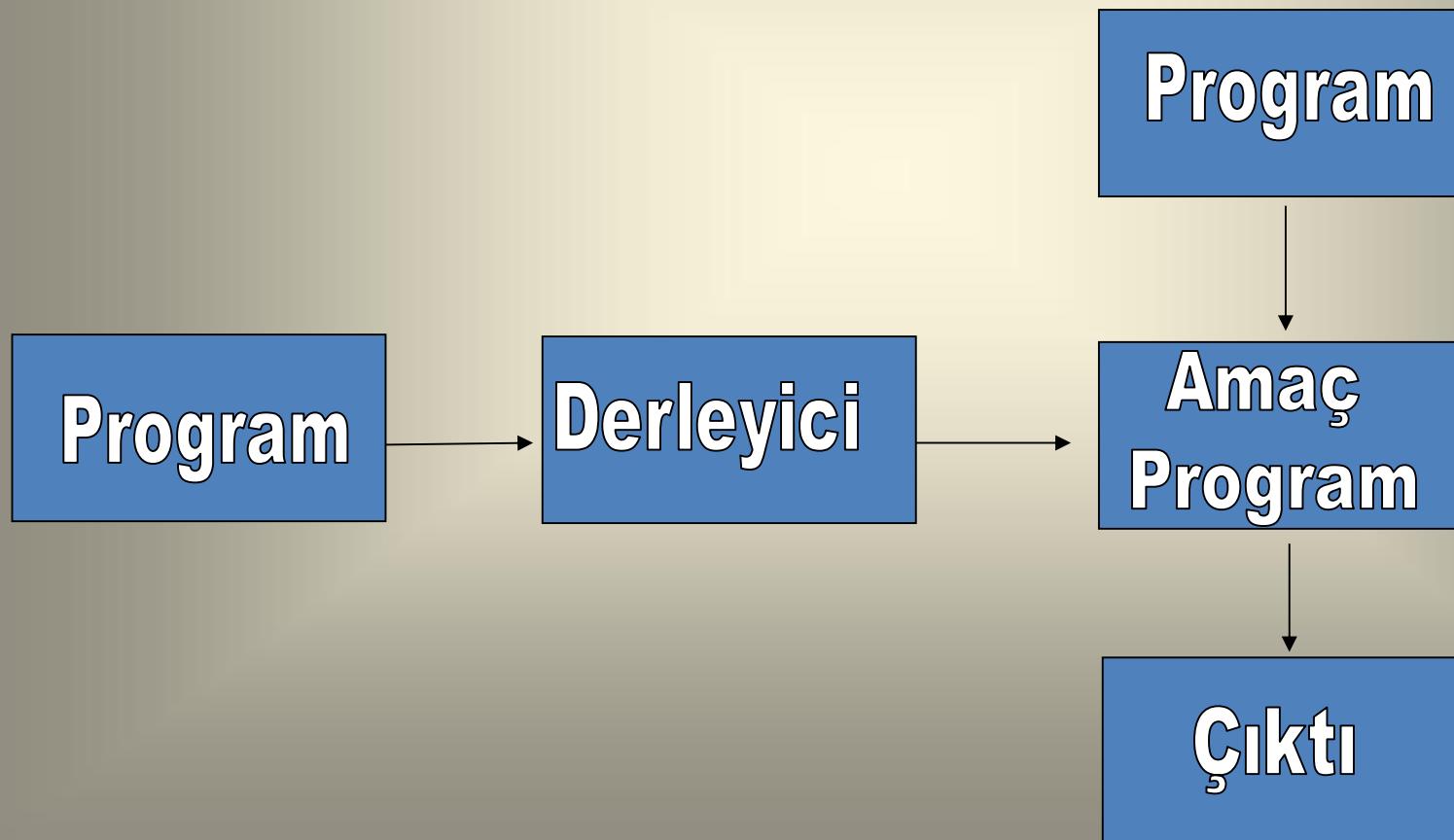
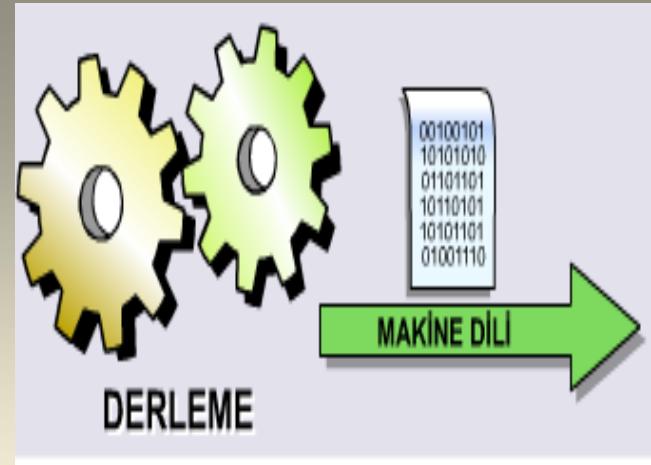
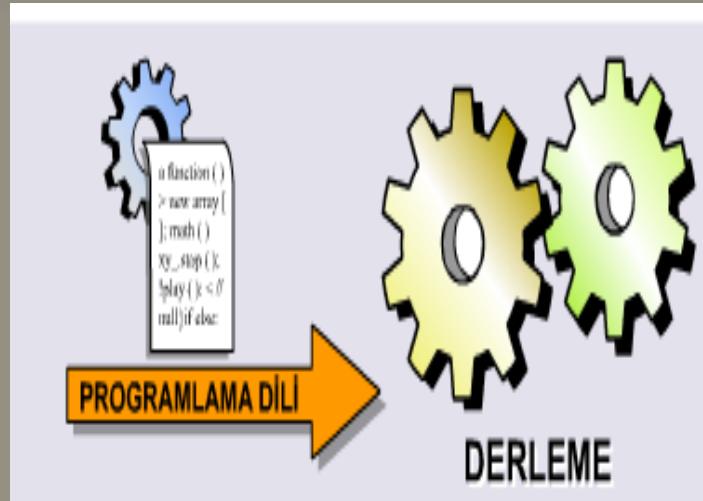


```
00101010101010  
10101011111010  
11101010101110  
00101010101010  
...  
.
```

```
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.
```

Derleme

```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c  
00401825 10820008 0064082a 10200003 00000000 10000002 00832023  
00641823 1483ffff 0064082a 0c1002b2 00000000 8fbf0014 27bd0020  
03e00008 00001025
```



Programlama Dili Nedir?

Makine Dili

- Bir programlama dilinin bilgisayar tarafından anlaşılması için, o dilin sözdiziminin ve anlamının makine diline çevrilmesi gereklidir.
- **Makine dili**, bir bilgisayarın doğrudan anladığı gösterim olup, bilgisayarların ana dili olarak nitelenebilir.
- Programlama dillerinin makine diline çevrilmesi, derleme veya yorumlama yöntemleriyle sağlanır.

Neden Farklı Programlama Dilleri? (1)

- Programlama dillerinin zaman içinde gelişimidir.
- Araştırmacıların programlama dillerinde daha iyi tasarımlar yapmaları sonucunda gelişme olmaktadır.
- İlk programlama dilleri 1950'lerin başlarında geliştirilmiştir.
- Bugünkü programlama dilleri ile karşılaştırıldıklarında oldukça zayıf bir dil tasarıımına sahiplerdi.
- Modern programlama dilleri ilk programlama dillerinden çok farklı özelliklere sahip olacak şekilde geliştirilmiştir.

Neden Farklı Programlama Dilleri? (1)

- Değişik programlama gereksinimleri ortaya çıkmıştır.
- Gereksinimlere göre, birbirlerinden çok farklı özelliklere sahip olan çok sayıda programlama dili geliştirilmiştir.
- Bilgisayarlarda kullanılan programlar genel olarak “sistem programları” ve “uygulama programları” olmak üzere ikiye ayrılmaktadır.
- Bazı programlama dilleri sistem programları, bazıları ise uygulama programları yazmak üzere tasarlanmıştır.
- Farklı uygulama türleri bulunduğuundan, bu uygulamalara cevap verebilecek çeşitlilikte programlama dilleri de geliştirilmiştir.
- Bir iş için uygun olan programlama dili, diğer iş için uygun olmayabilmektedir. Bilgisayarlar birbirinden farklı amaçlar için kullanıldığından, çok sayıda programlama dili geliştirilmiştir.

0.3. Programlama Dillerini Düzeylere Ayırmak

- Programlama dilleri, düşük düzeyli ve yüksek düzeyli olarak ikiye ayrılır.

Yüksek Düzeyli Programlama Dilleri

Yüksek düzeyli diller, bilgisayar donanımına bağımlı olmayan özellikteki dillerdir. Yüksek düzeyli diller, okunabilir bir gösterim, makine bağımsızlığı ve program kütüphaneleri sağladıkları için, makine dili ve birleştirici dilinin yerlerini almışlardır. Bu ders kapsamında sadece yüksek düzeyli programlama dilleri incelenecektir.

Düşük Düzeyli Programlama Dilleri

Düşük düzeyli programlama dillerinde, dilin özellikleri bilgisayar donanımına bağımlıdır. Düşük düzeyli programlama dilleri, makine dili ve birleştirici dilinden oluşur. Sadece 0 ve 1'lerden oluşan makine dili, insanlar tarafından anlaşılması güçmasına karşın, bilgisayarın ana dili özelliğini taşımaktadır. Birleştirici (Assembly) dilinde ise makine dilinden farklı olarak, makine işlemlerinin, değerlerin ve bellek yerleşimlerinin yerini, isimler ve semboller almıştır. Böylece, komutların okunabilirliği artmıştır.

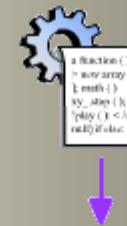
Örneğin;

ADD R1, 28

komutu R1 isimli kayıtçidakı değere 28 değerinin eklenmesini belirtmektedir.

0.4. Dil Çevrimi

- Yüksek düzeyli dilde yazılmış bir programın, bilgisayarda çalıştırılabilmesi için programın, o bilgisayarın makine diline çevrilmesi gereklidir.
- Bir programlama dili komutlarının makine diline çevrimini gerçekleştiren yazılımlara **dil çevirici yazılımlar** denir.
- Dil çevirici yazılımların oluşturulması ise **dilin gerçekleştirimi** olarak adlandırılır.
- Dil çevrimi için **derleme** ve **yorumlama** olarak adlandırılan iki temel yöntem vardır.

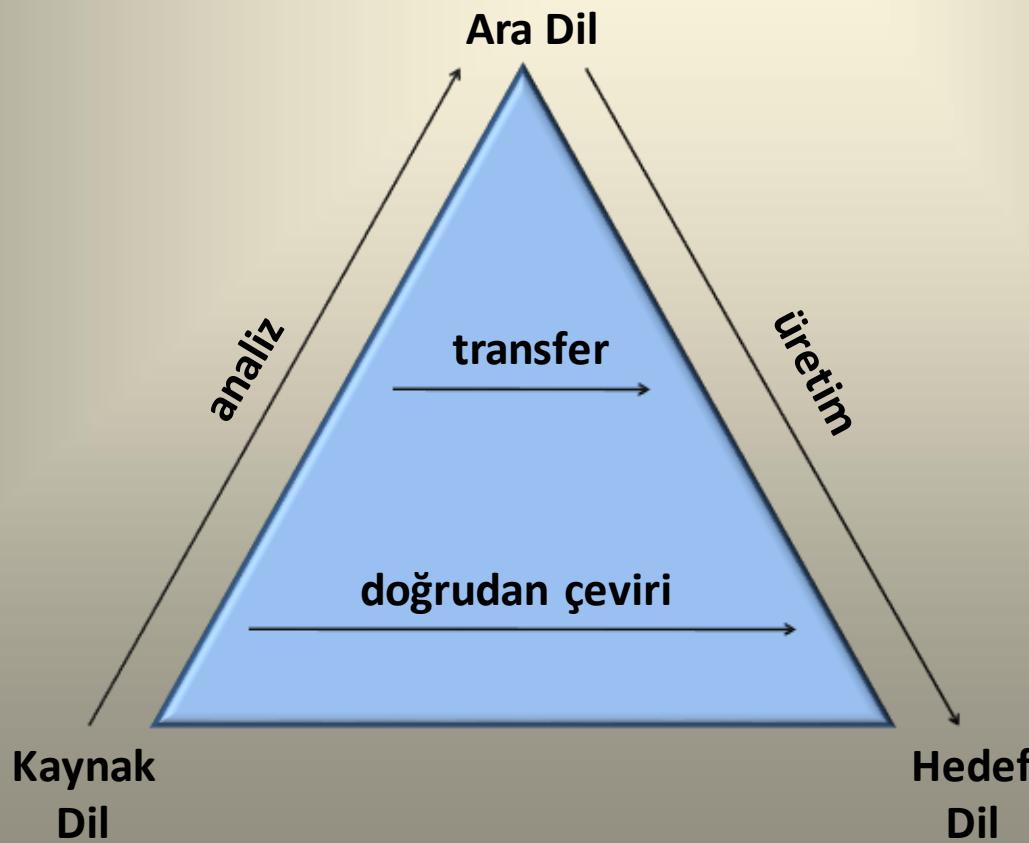


Dil Çevirici Yazılımlar
(Derleme / Yorumlama)



0.4. Dil Çevrimi

Otomatik Çeviri Paradigması



Yorumlayıcılar (Interpreters)

- Yorumlayıcılar, bir programı çok az ya da hiç, çalışma öncesi işleme (pre-processing) sokmadan olduğu gibi çalıştırırlar.
- **Yorumlayıcı**, bir programın her deyimini birer birer makine diline çevirir ve o deyimle ilgili bir altprogram çağrıarak deyimin çalıştırılmasını sağlar.
- Yani **yorumlama** yaklaşımı, bir deyimin makine diline çevrimini ve o deyimin bilgisayarda çalıştırılmasını birleştirmektedir.

Yorumlayıcılar (Interpreters)

- Yorumlayıcılar, dildeki her olası işlem için makine dilinde yazılmış bir altprogram içerirler.
- Bir programın yorumlanması, var olan altprogramların uygun sırada çağrılması ile gerçekleşir.
- Bir yorumlayıcının çalışması; *1. Sıradaki deyimi al, 2.Yapılacak işlemleri belirle, 3.İşlemleri gerçekleştir* adımlarının programdaki deyimler sona erene kadar yinelenmesinden oluşur.

Yorumlayıcılar

Girdi Verileri

Yorumlayıcı Yazılım

1. Sıradaki deyimi al.
2. Yapılacak işlemleri belirle.
3. İşlemleri gerçekleştir.

Çıktılar

```
> new array [ ]; math ( )  
xy_stop ( );  
play ( ) < />  
null ) if else;
```

Program

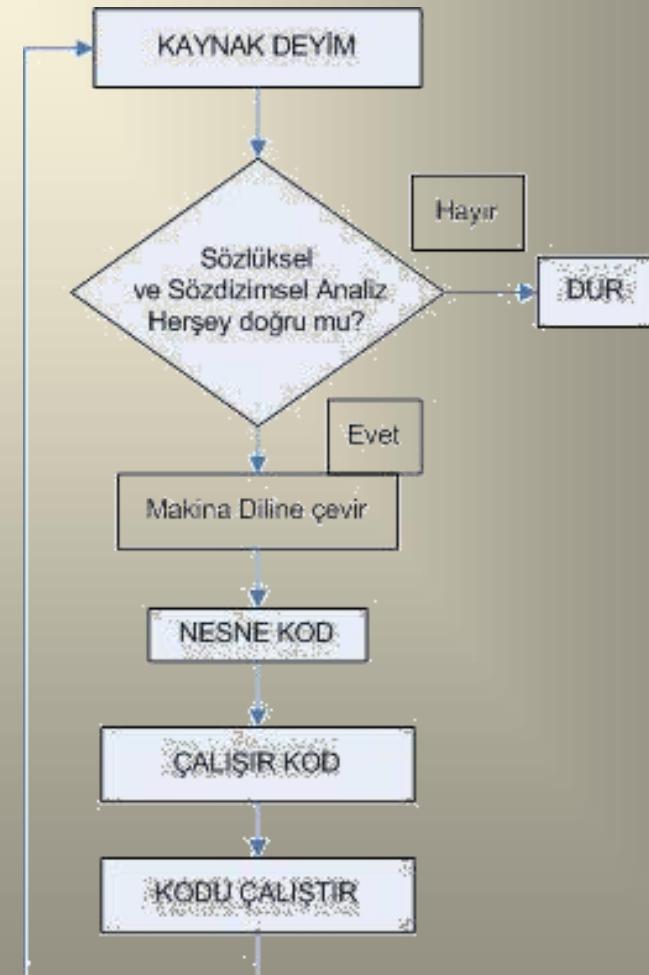
Program
Veri

Yorumlayıcı

Çıktı

Yorumlayıcının Genel Çalışma Şekli

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



Derleme Yaklaşımı ile Dil Çevrimi

- Bir **derleyici**, bir programlama dilinde yazılmış bir program için o programa eşdeğer olan makine dilinde bir program oluşturur.

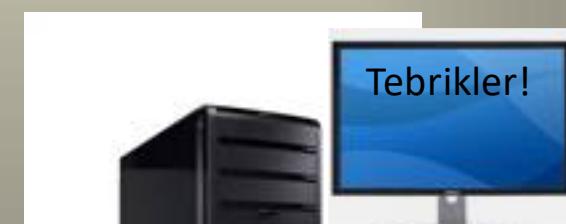
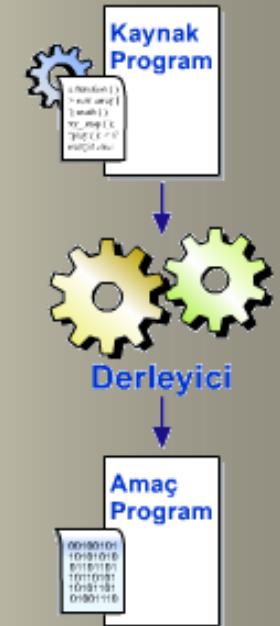
```
program.c
while (c != 'x')
{
    if (c == 'a' || c == 'e' || c == 'i')
        printf("Congrats!");
    else
        if (c != 'x')
            printf("You Loser!");
}
```

Compiler

gcc -o prog program.c

program

10000010010110100100101.....

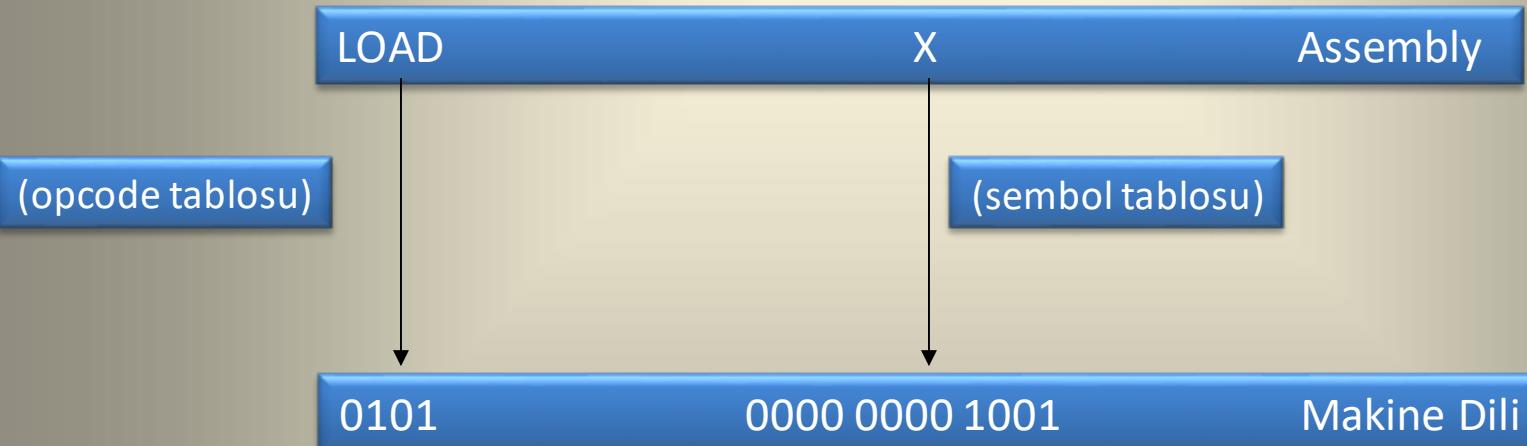


Derleme Sürecine Genel Bir Bakış

- Derleme sürecinin başlangıcında derleyiciye verilen yüksek düzeyli bir programlama dili deyimlerini içeren programa, **kaynak (source) program**, derleme sürecinin sonucunda oluşan makine dilindeki programa ise **amaç (object) program** adı verilir.
- Derleyicinin çalışması sırasında geçen zamana **derleme zamanı (compile time)** denir. Programın çalıştırılması, derleme sürecinden bağımsız olup, amaç programının çalıştırılmasından oluşur. Amaç programlarının çalışması sırasında geçen zamana **çalışma zamanı (run time)** adı verilir.

Derleme

Assembler (bir çeşit derleyici)

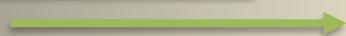


Bire-bir çeviri

Derleme

Derleyici (yüksek seviye dil çevirisici)

$a = b + c - d;$



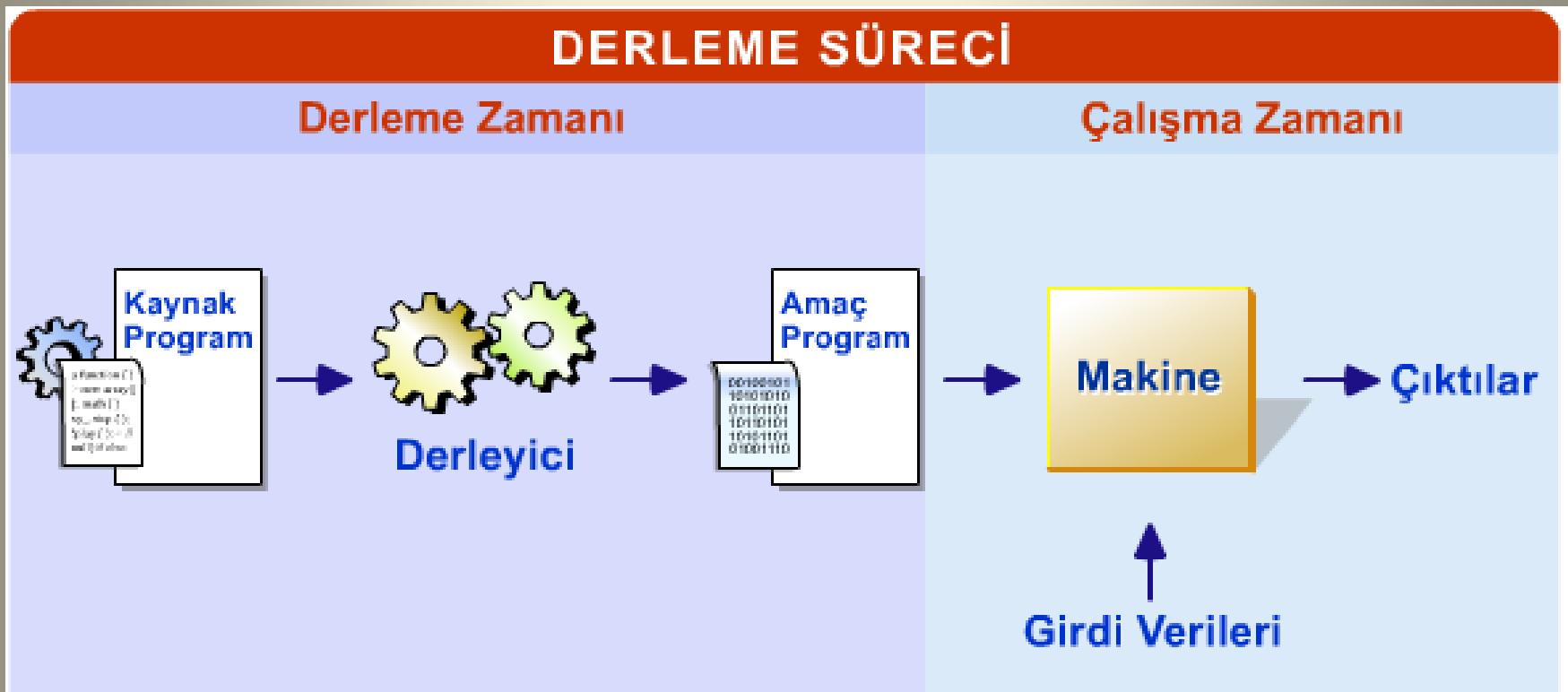
0101 00001110001	LOAD B
0111 00001110010	ADD C
0110 00001110011	SUBTRACT D
0100 00001110100	STORE A



0101 00001110001 0111 00001110010.....

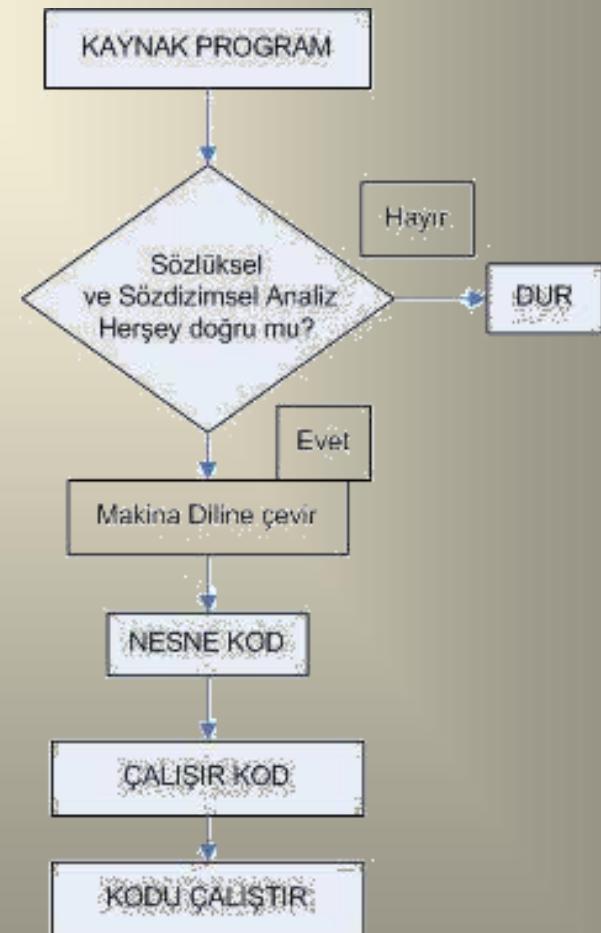
Birden-çoğa çeviri

Derleme Sürecine Genel Bir Bakış



Derleyicinin Genel Çalışma Şekli

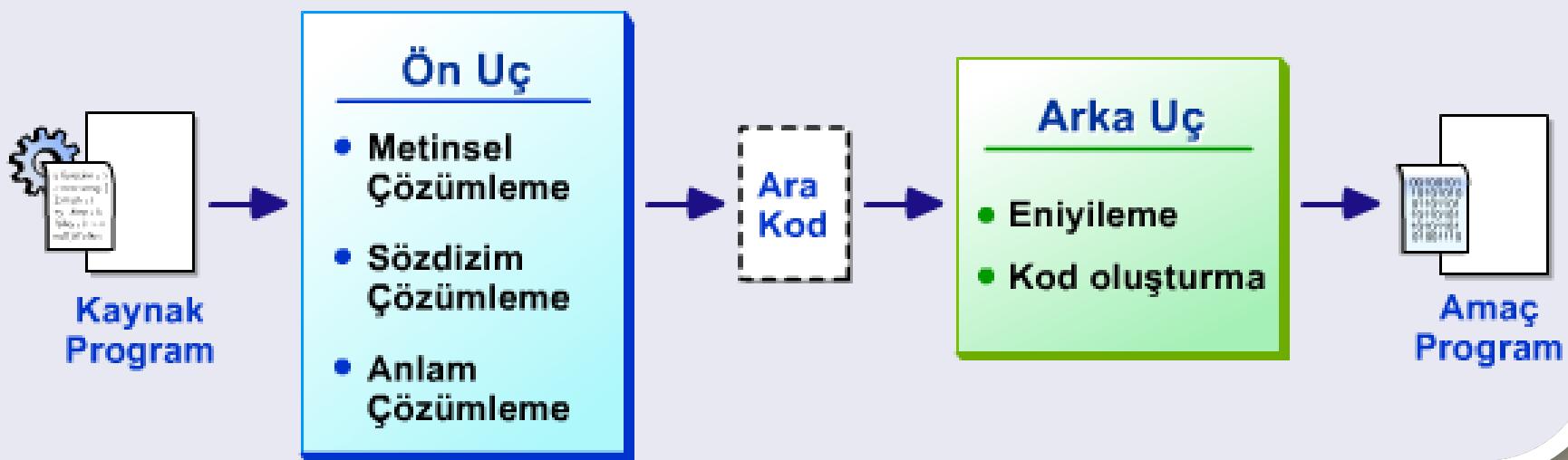
- Tüm programı bir kerede denetler.
- Sözlüksel ve sözdizimsel hataları bulur.
- Hata yok ise, programı nesne koda çevirir.
- Nesne kod daha sonra çalışabilir koda çevrilir



Derleme Sürecinin Aşamaları

- Derleyicinin **ön-ucu (*front-end*)**, bir programın sözdizimini ve anlamını dilin kurallarına göre inceleyerek çözümler ve bir ara kod oluşturur.
- Bu süreçte, metinsel çözümleme sonucunda token dizisi, sözdizim çözümlemesinin sonucunda ayrıştırma ağacı ve anlam çözümlemenin sonucunda da bir ara kod ile ifade edilen soyut program oluşturulur.
- Derleyicilerin **arka-ucu (*back-end*)** ise, programın ara kod gösterimine eniyileme uygulayarak, amaç programı, yani makine kodunu oluşturur.

Derleme Sürecinin Aşamaları



Metinsel (Lexical) Analiz

- Bir derleyicinin ön ucunda yer alan **metinsel çözümleyici** (*lexical analyzer*), bir kaynak programı bir dizi *token*'a çevirir.



Metinsel (Lexical) Analiz

- Token sözdizimsel bir kategori belirtir.
- Bu kategoriler doğal diller için “isim”, “sifat”, “fiil” vb. olabilirken, programlama dilleri için “matematiksel sembol” veya “anahtar kelime” gibi alt ifadeler olurlar.
- Token belirten bir alt karakter katarı lexeme olarak adlandırılır.
- Programlama dili içinde kullanılması mümkün olan tüm lexemeler tanımlanırken örüntüler (pattern) kullanılır.
- Örüntüler düzenli ifadeler (regular expressions) kullanılarak tanımlanır.
- Bu aşamada düzenli ifadelerin kullanılmasının sebebi işlenen karakter katarı içinde kabul edilebilir bir token bulabilmenin bazen geriye dönük arama da gerektirebilmesidir.

Metinsel (Lexical) Analiz-Sembol Tablosu

- Derleme sürecinde programdaki her tanımlayıcı için bir eleman içeren **sembol tablosu** oluşturulur. Sembol tablosu, derleme sürecindeki çeşitli aşamalarda kullanılır ve güncellenir.
- Bir tanımlayıcı kaynak programda ilk kez bulunduğuanda, o tanımlayıcı için sembol tablosunda bir eleman oluşturulur. Aynı tanımlayıcının daha sonraki kullanımları için ilgili *token*, aynı sembol tablosu elemanına başvuru içerir.
- Metinsel çözümleme aşamasının sonunda, programdaki *token*'lar ve her *token*'ın özelliklerinin tutulduğu sembol tablosu elemanına işaret edilen göstergeleri içeren *token dizisi* oluşturulur.
- Lexical analiz işleminin çıktısı olan tokenlar bir sonraki aşama olan sözdizimsel (sentaks) analiz bileşenine aktarılır

Sözcüksel veya Metinsel (Lexical) Analiz

- Aşağıdaki tabloda görülen örnekte, bir Pascal deyimi için metinsel çözümünün sonucunda oluşan *token* dizisi görülmektedir.

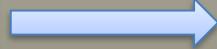
Örnek	
toplam: = değer + 10;	
toplam	(tanımlayıcı, gösterge 1)
:	(iki_nokta, nil)
=	(eşit, nil)
değer	(tanımlayıcı, gösterge 2)
+	(arti, nil)
10	(tamsayı, gösterge 3)
;	(noktalıvirgül, nil)

"toplam:= değer + 10 ;" deyimi için token dizisi.

Sözcüksel veya Metinsel (Lexical) Analiz

- Bir lexical analiz aracı şu 3 şeyi yapabilmelidir:
 1. Bütün boşlukları ve açıklamaları temizlemeli,
 2. Karakter katarı içindeki tüm tokenlar bulunmalı,
 3. Bulunan token için lexeme ve bulunduğu satır numarası gibi özellikler döndürülmelidir.

toplam=3+2 ;



Lexeme	Token
toplamp	IDENTIFIER
=	ASSIGN_OP
3	NUMBER
+	ADD_OP
2	NUMBER
;	SEMICOLON

Sözcüksel veya Metinsel (Lexical) Analiz

Düzenli İfadeler

- Sonlu durum otomatları (FSA) ile tanımlanabilen dilleri ifade etmede kullanılırlar.
- Eğer A bir düzenli ifade ise, $L(A)$ bu ifade ile tanımlanan dildir.
- $L(\text{"if"} \mid \text{"then"} \mid \text{"else"})$ dili sadece "if", "then" ve "else" ifadelerini tanıyalı bilen bir dil belirtir.
- Lexical analiz işleminde tarayıcı (scanner), kaynak kodun içinde önceden düzenli ifadelerle tanımlanmış anahtar kelimeleri arar ve bunların token tiplerini belirler.

Sözcüksel veya Metinsel (Lexical) Analiz

Düzenli İfadelerin Tasarımı

- Tanımladığımız düzenli ifadeler dil içerisindeki tüm ifadeleri kapsamalıdır. Bu sebeple şu durumlar dikkate alınmalıdır:
 1. Bir düzenli ifade özel bir token tipinin tüm durumlarını ifade etmelidir.
 2. Belirsizliği önlemek için *en uzun* olan eşleme seçilir.
 3. Eş uzunluklu belirsizlik durumlarında *ilk* eşleme seçilir.
- n adet farklı token tipi olduğu varsayıldığında n farklı düzenli ifade kullanılmalıdır. Bir token tipini tanıyan düzenli ifade R_n ile gösterilirse, tüm tokenları tanıyan düzenli ifade topluluğu R şöyle ifade edilir:

$$R = R_1 / R_2 / \dots / R_n$$

Sözcüksel Analiz Şekli

sonuc = a + b * 10

Tarayıcı

DÜZENLİ İFADELER

[\t\n]*	→ boşluklar
[a-z][A-zA-Z0-9]*	→ identifier
[0-9]+"	→ number

Lexeme

sonuc

=

a

+

b

*

10

Token

IDENTIFIER

"="

IDENTIFIER

"+"

IDENTIFIER

"*"

NUMBER

identifier

"="

identifier

"+"

identifier

"**"

number

sonuc

=

a

+

b

*

10

Sözcüksel veya Metinsel (Lexical) Analiz

```
/*Metinsel çözümlemenin gerçekleştirimi*/
int lex() {
    getChar();
    switch (charClass) {
        /* Tanımlayıcıları ve özel amaçlı kelimeleri ayırtırma*/
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT)
            {
                addChar();
                getChar();
            }
            return lookup(lexeme);
            break;
        ...
    }
}
```

Sözcüksel veya Metinsel (Lexical) Analiz

...

```
/*Tamsayı sözcüklerini ayırtırma*/
case DIGIT:
    addChar();
    getChar();
    while (charClass == DIGIT) {
        addChar();
        getChar();
    }
    return INT_LIT;
    break;
} /* switch'in sonu */
} /* lex fonksiyonunun sonu */
```

Sözcüksel veya Metinsel (Lexical) Analiz

Aşağıdaki kod fragmanını düşünün:

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

Lexical analizci bunu karakterler stringi olarak okur:

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

Lexical analiz stringi tokenlara ayırır.

Sözcüksel veya Metinsel (Lexical) Analiz

Token ve Token Değeri

"y = 31 + 28*foo"

Lexical
Analizci

karakter alır

<id, "y"> <=,> <int, 31> <+,> <int, 28> <*,> <id, "foo">
token
token değeri

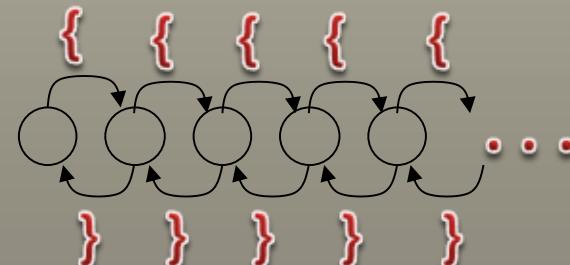
Tokenları alır
(her seferinde bir tane)

Syntax Analizci

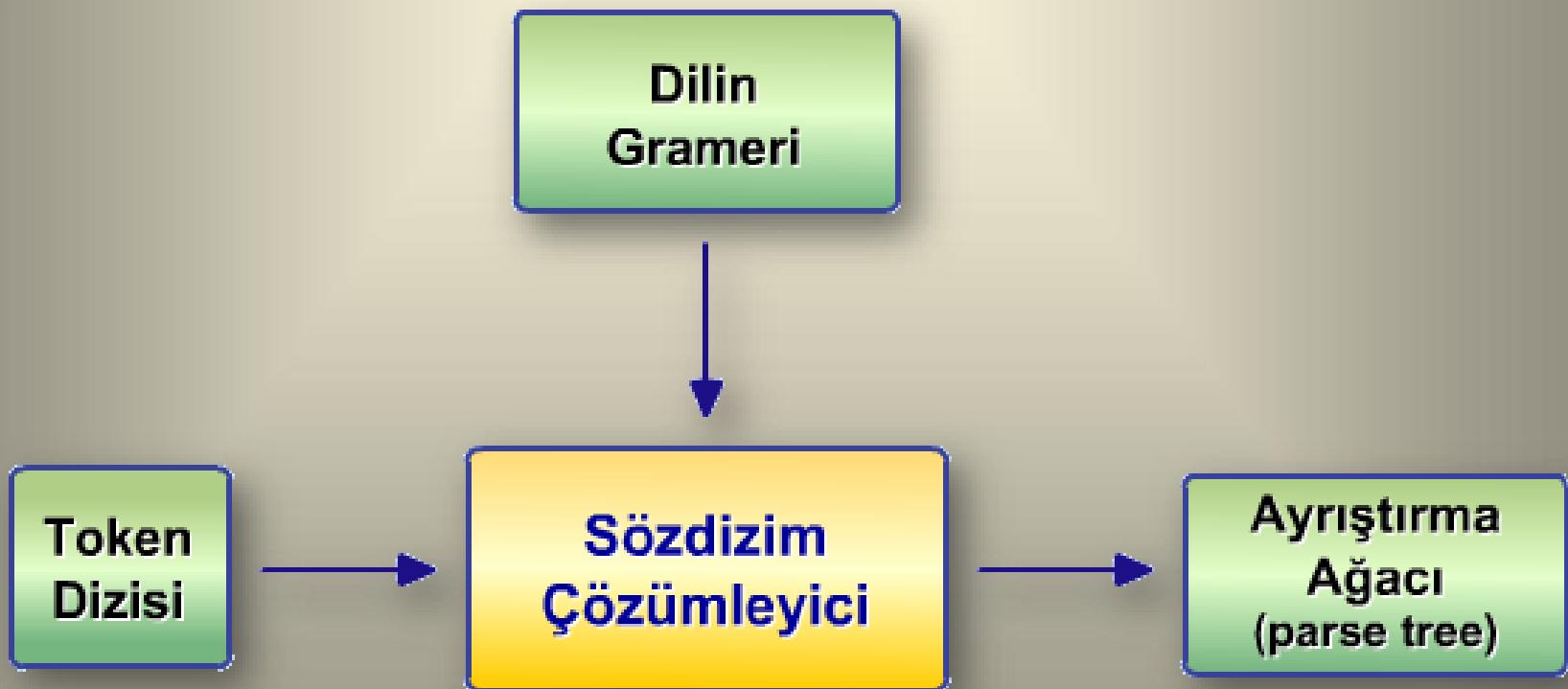
Sözdizim (Syntax) Çözümleme

- Sözdizim çözümleme, bir program için metinsel çözümleme sonucunda oluşturulmuş *token* dizisini ve o programlama dilinin gramerini kullanarak, **ayrıştırma ağacı** (*parse tree*) oluşturmayı amaçlar.
- Böylece, metinsel çözümlemeden geçmiş bir programın sözdizim olarak doğru olup olmadığı belirlenir. Sözdizim çözümleyici, aynı zamanda **ayrıştırıcı** (*parser*) olarak adlandırılır.
- Sözdizim çözümleyiciler, derleyiciyi gerçekleştirenler tarafından dilin grameri kullanılarak yazılır.
- Sözlüksel analiz için düzenli ifadeleri kullanan lex'ten faydalandık.
- Düzenli dillerin çözebildiği yapıların karmaşıklığı sınırlı olduğu için sözdizim analizinde kullanılmaya yetmez.

{} {} {{}} { } }

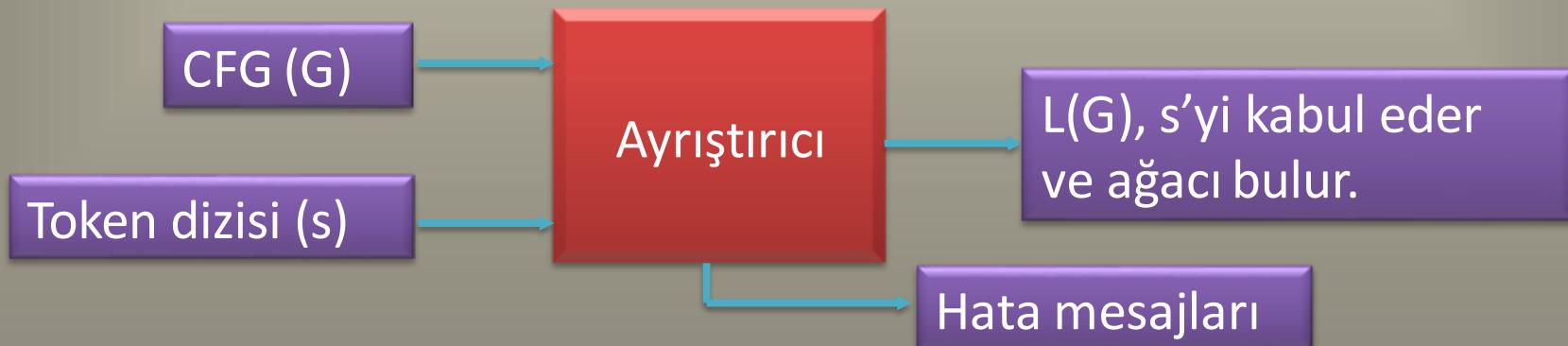


Sözdizim (Syntax) Çözümleme



Sözdizim (Syntax) Çözümleme

- Verilen bir program için ayrıştırıcının (parser) amacı:
 - Tüm sözdizimsel hataları bulmak; her hata için iyileştirici mesajlar yayımlamak ve gerekirse düzeltmeler yapmak.
 - Bir ayrıştırma ağacı oluşturmak



Sözdizim (Syntax) Çözümleme

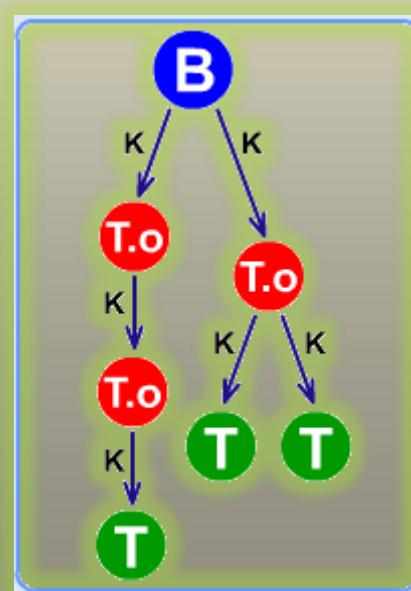
Bağlam-Bağımsız Gramerler (CFG)

- CFG'ler birçok programlama dilini tanımda yeterlidir.
- 4 temel bileşenden oluşurlar:
 1. Terminal semboller (a, b)
 2. Terminal olmayan semboller (T)
 3. Başlangıç simbolü (S)
 4. Üretim kuralları
- Karşılıklı parantezleri tanıyan bir CFG söyledir:
 - $S \rightarrow (S)S$
 - $S \rightarrow \epsilon$
 - 1 terminal olmayan: S
 - 2 terminal: "(", ")"
 - Başlangıç simbolü: S
 - 2 üretim kuralı

$$\begin{aligned}S &\rightarrow aSa \\S &\rightarrow T \\T &\rightarrow bTb \\T &\rightarrow \epsilon\end{aligned}$$

Sözdizim (Syntax) Çözümleme -Ayrıştırma Ağacı Oluşturma

- Bir karakter dizgi için ayrıştırma ağacı oluşturulmasında, **yukarıdan aşağıya** veya **aşağıdan yukarıya** olmak üzere iki teknik uygulanabilir.



Sözdizim (Syntax) Çözümleme -Ayrıştırma Ağacı Oluşturma

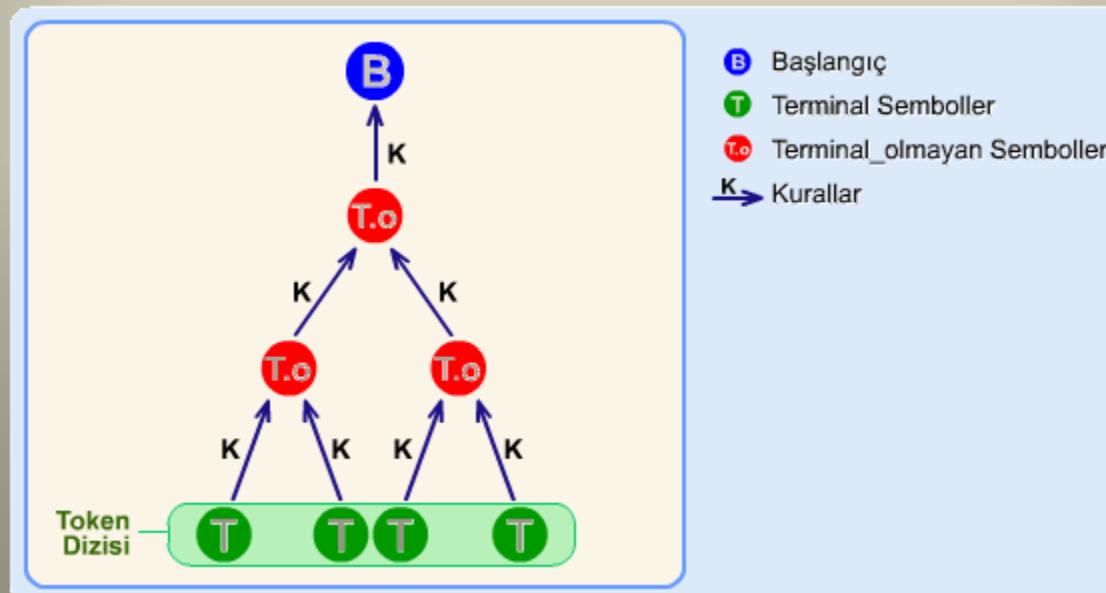
- Yukarıdan Aşağıya (Top Down) :
- Ayrıştırma ağacı kökten başlayarak oluşturulur.
- Yukarıdan aşağıya ayrıştırma ağacı oluşturma tekniğinde ilk adım, o karakter dizginin amaç sembolün *token*'larıyla eşlenme olasılığı olan tanımlarından birisinin belirlenmesidir.
- Bu tanımlardan elde edilen terminal olmayanların tanımlarının belirlenmesiyle işleme devam edilir. İşlem bu şekilde, tanımlanmamış terminal olmayan kalmayınca kadar devam eder.

Sözdizim (Syntax) Çözümleme -Ayrıştırma Ağacı Oluşturma

- Bazen türetme sürecinde ölü nokta olarak adlandırılan ve çözümleyicinin **geri izleme** (*backtracking*) yapmasını gerektiren bir noktaya gelinebilir.
- Geri izleme, en yakın tanımlamaya geri dönülerek, o noktada bulunan kullanılmamış tanımlardan birisinin türetme sürecinde kullanılması demektir.

Sözdizim (Syntax) Çözümleme -Ayrıştırma Ağacı Oluşturma

- **Aşağıdan Yukarıya (Bottom up)** : Aşağıdan yukarıya ayrıştırma ağacı oluşturma tekniğinde ise, işleme bir dizi token ile başlanır ve bunların olası terminal olmayanları belirlenerek, amaç sembole ulaşıncaya kadar işleme devam edilir.



Sözdizim Analizi

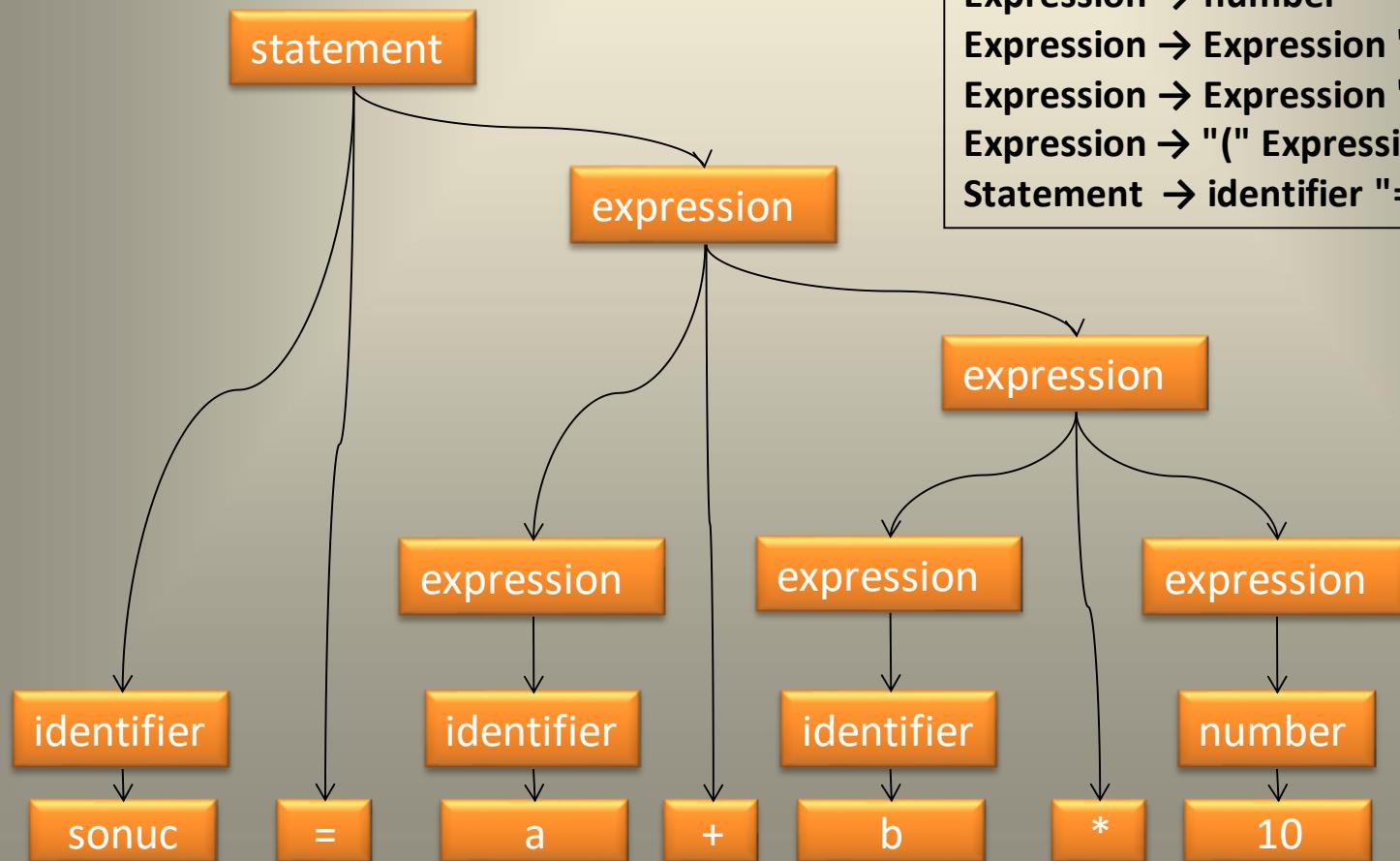


CFG \longrightarrow BNF (Backus-Naur Form)

```
 $\Sigma = \{a,b\}$   
 $N = \{S\}$   
 $R = \{S \rightarrow aSb, S \rightarrow \epsilon\}$ 
```

```
select_command ::= "select" [ "all" | "distinct" ] ( "*" |  
    (displayed_column { "," displayed_column } ) ) "from" (   
    selected_table { "," selected_table } ) [ "where"   
    condition ] { connect_clause } { group_clause } {   
    set_clause } { order_clause } { update_clause }
```

Ayrıştırma



Expression → identifier
Expression → number
Expression → Expression "+" Expression
Expression → Expression "*" Expression
Expression → "(" Expression ")"
Statement → identifier "=" Expression

Sözdizim Analiz- Tanıma- Bottom Up

Op = '+' | '-' | '*' | '/'

Int = [0-9] +

Open = (

Close =)

1) *Start* → *Expr*

2) *Expr* → *Expr Op Expr*

3) *Expr* → *Int*

4) *Expr* → *Open Expr Close*

(2 - 1) + 1

Open Int Op Int Close Op Int

Open *Expr* Op *Expr* Close Op *Expr*

Open *Expr* Close Op *Expr*

Expr Op *Expr*

Expr

Start

Sözdizim Analiz- Üretim-Top Down

1) $Start \rightarrow Expr$

2) $Expr \rightarrow Expr \text{ Op } Expr$

3) $Expr \rightarrow Int$

4) $Expr \rightarrow Open \text{ } Expr \text{ Close}$

Start

Expr

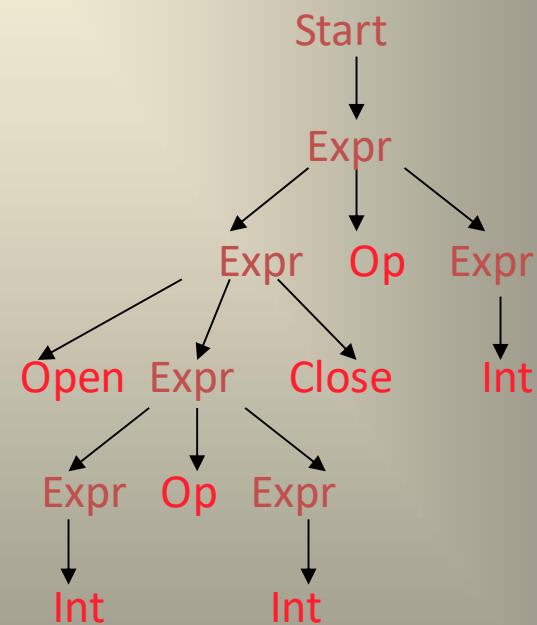
Expr Op Expr

Open Expr Close Op Expr

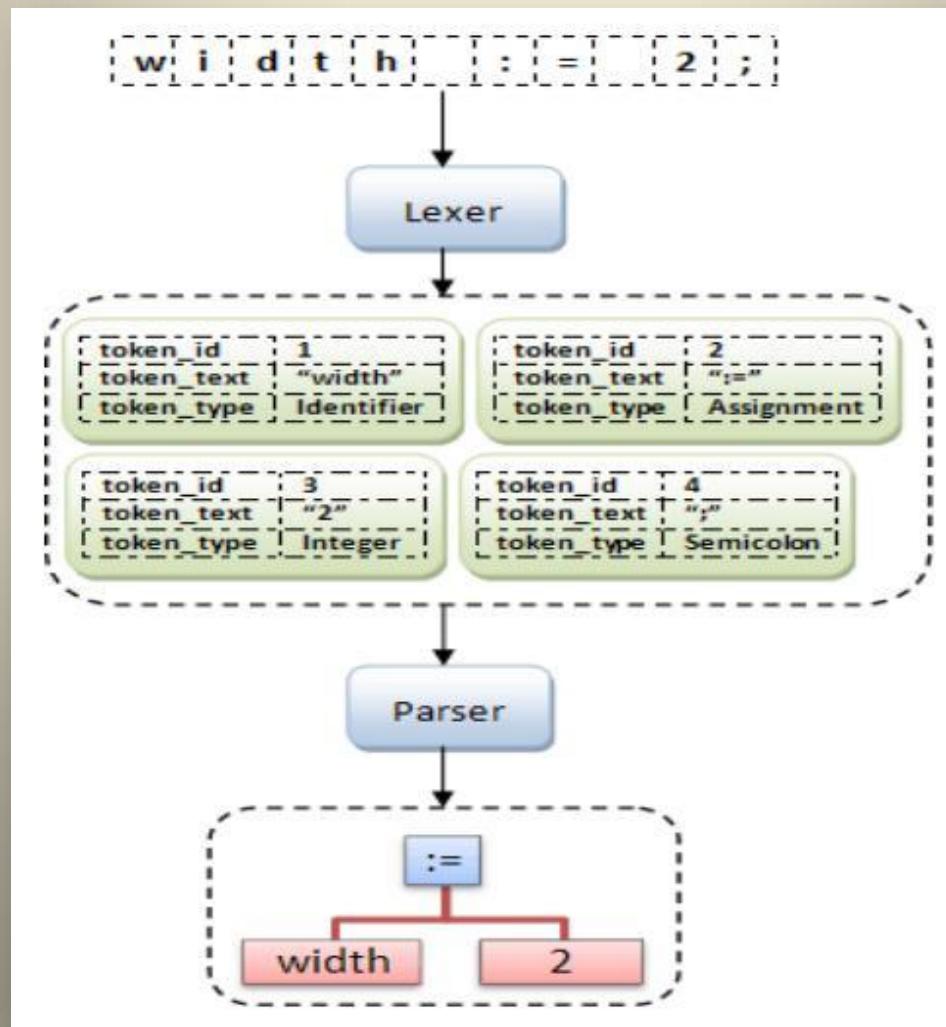
Open Expr Op Expr Close Op Expr

Open Int Op Int Close Op Int

$(2 - 1) + 1$



Sözcüksel ve Sözdizim Analiz Şekli



Belirsizlik

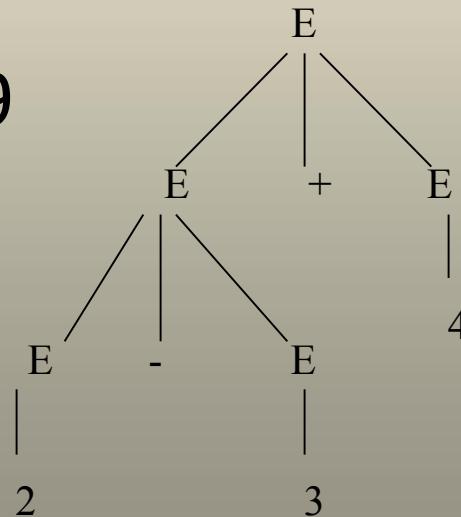
Aynı string için iki (ya da daha fazla) ayrıştırma ağacı

$$E \Rightarrow E + E$$

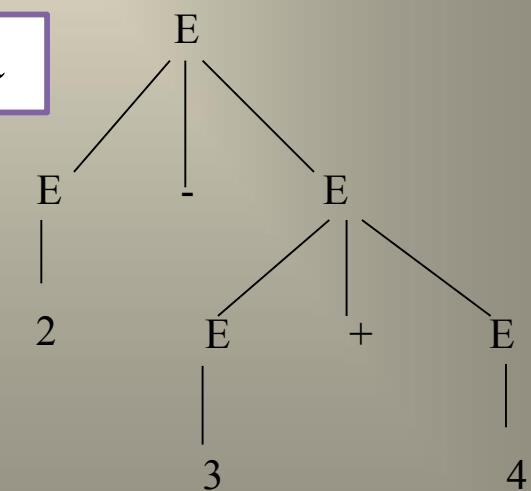
$$E \Rightarrow E - E$$

$$E \Rightarrow 0 \mid \dots \mid 9$$

2 – 3 + 4



ya da



- İki türetme:

$$E \Rightarrow E + E$$

$$\Rightarrow E - E + E$$

$$\Rightarrow 2 - E + E$$

$$\Rightarrow 2 - 3 + E$$

$$\Rightarrow 2 - 3 + 4$$

$$E \Rightarrow E - E$$

$$\Rightarrow 2 - E$$

$$\Rightarrow 2 - E + E$$

$$\Rightarrow 2 - 3 + E$$

$$\Rightarrow 2 - 3 + 4$$

Belirsizliği Giderme

- Belirsiz bir gramer bazen belirli hale getirilebilir:

$$E \Rightarrow E + T \mid E - T \mid T$$

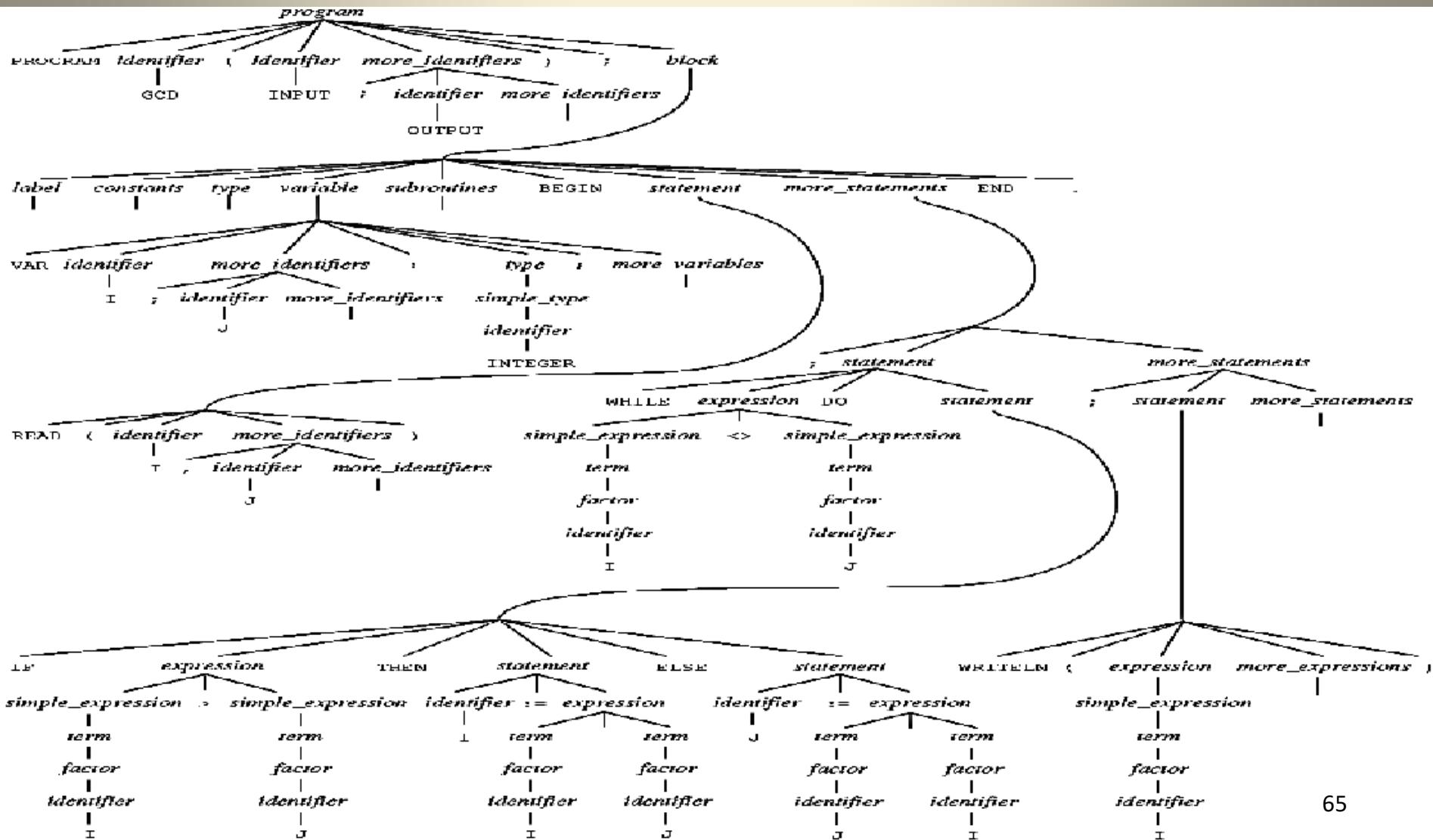
$$T \Rightarrow 0 \mid \dots \mid 9$$

Örnek analiz

- Pascal program örneğine ait ayrıştırma ağacı

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

Örnek analiz



Anlam (Semantics) Çözümleme

- **Anlam çözümleme**, kaynak program için sözdizim çözümleme sırasında oluşturulmuş ayrıştırma ağacı kullanılarak, soyut bir programlama dilinde bir program oluşturulmasıdır.
- Anlam çözümleyici, her deyim için sözdizim çözümleyicisinin belirlediği sözdizime göre bir çevrim altyordamı çağrıır.



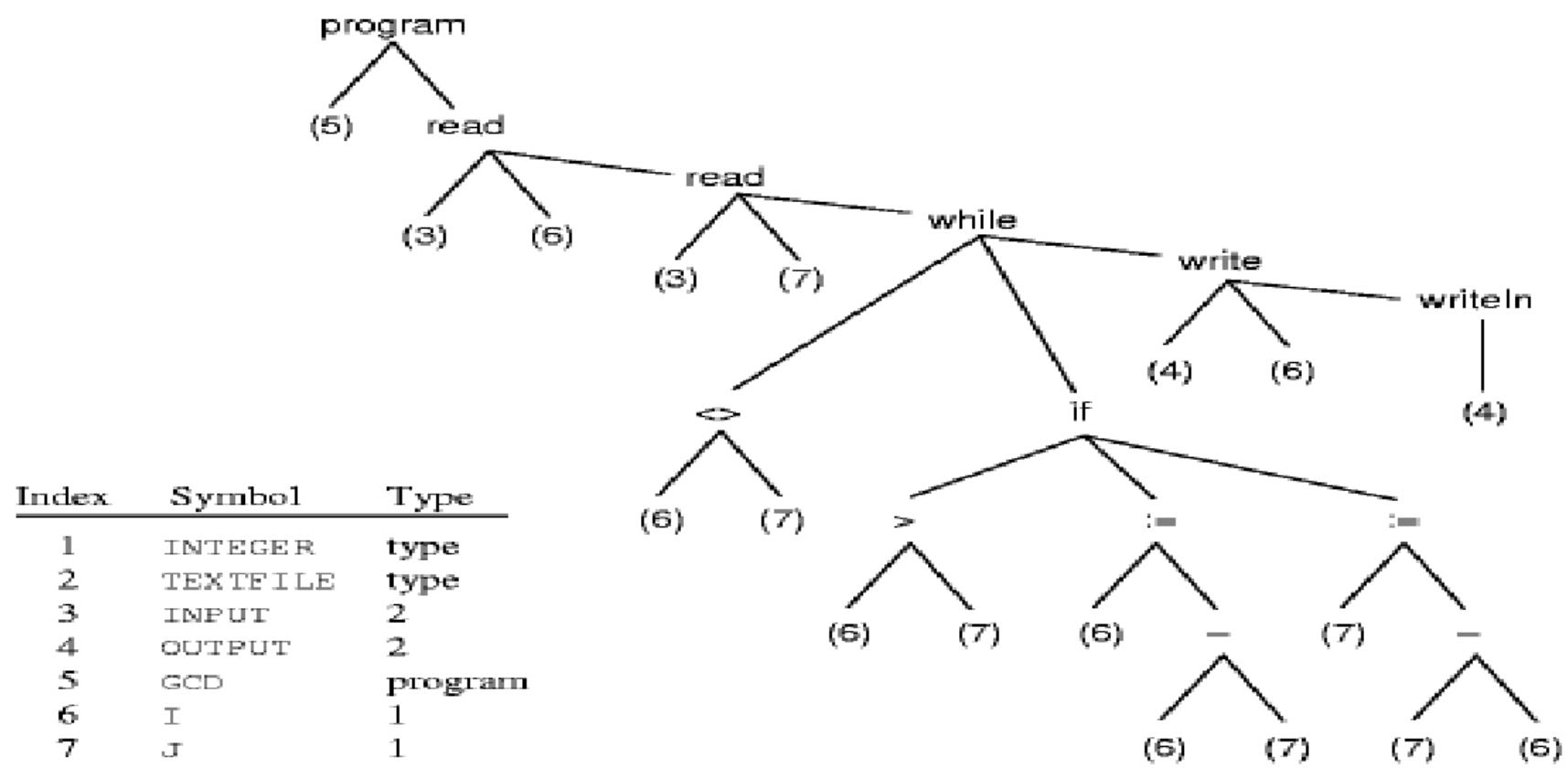
Anlam (Semantics) Çözümleme

- **Soyut Dil:** Anlam çözümleme sonucunda üretilen kod için kullanılan ara diller, genel olarak, üst düzeyli bir birleştirici diline benzerler.
- Bu soyut dil, kaynak dilin veri türleri ve işlemleriyle uyumlu olacak şekilde tasarlanmış, hayali bir makine için makine dili olup, derleyicinin kaynak ve amaç dilleri arasında bir ara adım oluşturur.

Örnek: Anlam Analizi

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

Örnek: Anlam Analizi



Kod Oluşturma

- Bir derleyicinin **arka ucu**, soyut dilde ifade edilen kodu alır ve belirli bir bilgisayar için makine kodunu oluşturur.
- Derleyicinin **ön ucu** programlama diline bağlı, **arka ucu** ise bilgisayara bağlıdır.



Kod Oluşturma

- Bir derleyici üreticisi bir programlama dili için, farklı bilgisayarlarda aynı ön ucu kullanan bir ara kodu alan farklı arka uçlar yazarak, derleyici aileleri üretebilir.
- Benzer şekilde bir bilgisayar üreticisi, bir bilgisayar için iyi bir arka uç oluşturuktan sonra, farklı kaynak dilleri ortak ara koda dönüştüren ön uçlar yazarak birçok dili destekleyebilir.

Optimizasyon (Eniyileme)

- Derleyicinin arka ucunda bulunan bir diğer bileşen, kodu daha etkin yapmaya çalışan eniyileyicidir.
- Programları otomatik olarak değiştirerek
 - Daha hızlı çalışmasını
 - Daha az bellek kaplamasını ve
 - Genel olarak kaynak kullanımında tutumlu davranışmayı sağlamaktadır
- Optimizasyon ile bir programın daha etkin olarak çalışacak bir eşdeğer programa dönüştürülmesi amaçlanır. Optimizasyon ikiye ayrılır;
 - Makine Bağımsız Optimizasyon
 - Makine Bağımlı Optimizasyon
- Kod optimizasyonu her ne kadar ara kod üretiminden sonra tek bir modül biçiminde gösterilmişse de sentaks analizinden sonra parse tree üzerinde ve hedef kod üretimi sırasında da ciddi optimizasyonlar yapılmaktadır. Fakat gerçekten de klasik optimizasyonların büyük bölümü ara kod üretiminden sonra gerçekleştirilir.

Optimizasyon - Makine Bağımsız Optimizasyon

- Makine bağımsız optimizasyon, kaynak programdan soyut programa kadar olan herhangi bir ara şekilde gerçekleştirilebilir.
- Örneğin, değerleri bilinen ifadelerin derleme zamanında hesaplanması, aynı hesaplamlarının optimizasyonu ve işlemcilerin eşdeğer ama daha hızlı olan işlemcilerle değiştirilmesi gibi işlemler, makine bağımsız optimizasyon örnekleridir.
- Aşağıdaki deyimde, $a+b$ nin iki kez hesaplanması yerine, bir kez hesaplandıktan sonra bir kayıtçida saklanması bir optimizasyon örneğidir.

Optimizasyon - Makine Bağımsız Optimizasyon

Örnek

sonuc= fonksiyon1 (a+b) + fonksiyon2 (a+b)

Eniyileştirilmemiş

sonuc= fonksiyon1 (a+b) + fonksiyon2 (a+b)

1.İşlem

2.İşlem

NOT: Yukarıdaki işlemde $a+b$ işlemi iki kez yapılır.

Eniyileştirilmiş

c = a+b

sonuc= fonksiyon1 (c) + fonksiyon2 (c)

NOT: Yukarıdaki işlemde $a+b$ işlemi bir kez yapılır. Böylece kod eniyileştirilmiş olur.

Optimizasyon - Makine Bağımlı Optimizasyon

- Makine bağımlı optimizasyon, çalışma süresini kısaltmak için makina dili özelliklerini kullanır. İkinin bir kuvveti ile çarpım işlemini, **bir kaydırma (*shift*)** işlemi ile değiştirmek veya bir ekleme işlemi **yerine artış (*increment*)** komutunu kullanmak, makine bağımlı optimizasyon örnekleridir.
- Optimizasyon sonucu, çalıştırılan deyimlerin sırasının kaynak koddaki deyim sıralamasından farklı olabilmesi, hata ayıklama (*debug*) işlemini güçlendirir. Bu nedenle, hata ayıklama sırasında optimizasyonu çalışmamak gereklidir.

Eniyileştirilmemiş	çarpım, ekle
Eniyileştirilmiş	<i>shift</i> işlemi artış komutu

Optimizasyon Temaları

- Kod optimizasyonu çeşitli tipik kategorilere ayrılarak ele alınabilir.
- Döngüler en önemli optimizasyon kaynağıdır. Yani döngülere yoğunlaşılması önemlidir, çünkü program yaşamının büyük bir çoğunluğu döngülerde geçer.
- Optimizasyon işlemi hız ve kaynak kullanımı ölçütlerine bağlı olarak yapılabilir. Yani kod daha hızlı çalışacak biçimde ya da daha az yer kaplayacak biçimde optimize edilebilir. Ancak hız optimizasyonu çok daha baskın olarak tercih edilmektedir.
 - Ölü Kod Eliminasyonu (Death Code Elimination)
 - Gereksiz Kodların Elimine Edilmesi
 - Ortak Alt İfadelerin Elimine Edilmesi (Common Subexpression Elimination)
 - Sabit İfadesi Yerleştirme (Constant Folding)
 - Sabit İfadelerinin Yaydırılması (Const Propagation)
 - Ortak Blok (Basic Block)
 - Göstericilerin Eliminasyonu
 - Inline Fonksiyon Açımı
 - Döngü Değişmezleri (Loop Invariants)
 - Döngü Açıımı (Loop Unrolling)
 - Döngü Ayırması (Loop Splitting)
 - Kod Üretili Aşamasında Yapılan Optimizasyonlar
 - Yazmaç Tahsisatı (Register Allocation)
 - Komut Çizelgelemesi (Instruction Scheduling)

Ölü Kod Eliminasyonu (Death Code Elimination)

```
void Foo(void)
{
    //*****
    return;
    a = 10;      → Ölü kod
    b = 20;      → Ölü kod
}
```

```
{
    int a = 10;

    //*****
}
```

```
if (0)
    ifade;

short a;

if (a > 150000) {
    //*****
}
```

Gereksiz Kodların Elimine Edilmesi

Derleyici akışsal analiz yaparak gereksiz kodları elemine etmeye çalışır. Örneğin:

```
if (ifade) {  
    /****/  
    ****/  
    ifadel;  
}  
else {  
    /***/  
    ***/  
    ifadel;  
}
```

Burada ifadel'in if dışına yerleştirilmesi eşdeğerliği bozmadı. Fakat gereksiz bir kod elimine edilmiş olur. Örneğin:

```
if (a)  
    return 1;  
else  
    return 0;
```

Burada yapılmak istenen şudur.

```
return a != 0;
```

Ortak Alt İfadelerin Elimine Edilmesi (Common Subexpression Elimination)

Peşi sıra ifadeler içerisindeki ortak alt ifadeler bir geçici değişkene yada yazmaca alınarak hız kazancı sağlanabilir. Örneğin:

```
a = x + y + z;  
b = x + y + k;  
c = x + y + l;
```

Burada $x + y$ ortak alt ifadedir. Dolayısıyla aslında aşağıdaki kod daha hızlı çalışır.

```
temp = x + y;  
a = temp + z;  
b = temp + k;  
c = temp + l;;
```

Burada gereksiz bir biçimde $x + y$ tekrar tekrar yapılmayabilir. Şüphesiz burada temp bir yazmaç olabilir. Böyle değerlendirme yapıldığında tek başına değişkenlerinde bir alt ifade biçiminde yorumlanabileceği görülür. Örneğin:

```
a = x;  
b = x;  
c = x;
```

Burada her defasında bellekteki x 'i yeniden çekmektense bir kere yazmaca çekip yazmaca atama yapmak daha hızlı çalışmaya yol açar.

Kritik kodlar dışında programcı için okunabilirlik çok daha önemli olmalıdır. Yani derleyicilerin zaten yapabildiği optimizasyonları kod içerisinde yapmak kodun okunabilirliğini azaltabilir. Yukarıdaki örnekte $x + y$ alt ifadesinin temp'e programcı tarafından yerleştirilmesi kötü bir tekniktir. Güçlü derleyicilerin hemen hepsi bu tür eliminasyonları yapabilmektedir.

Sabit İfadelerin Yerleştirme (Constant Folding)

Sabit ifadelerinin değerleri çalışma zamanına bırakılmadan derleme zamanında hesaplanabilir. Örneğin:

$x = 10 + y + 2;$

Derleyici bu kodu $x = 12 + y;$ biçimine dönüştürebilir.

Sabit İfadelerinin Yaydırılması (Const Propagation)

Bu optimizasyon temasında derleyici değiştmeyeceğini bildiği ifadelerin değerlerini ya baştan hesaplar ya da değişken yerine sabit kullanır. Sabit kullanımı da pek çok mimaride hız kazancı sağlamaktadır. Örneğin:

```
int x = 10 + 2;  
int y;  
  
y = x + 5;
```

Burada derleyici x içerisindeki ifadenin değişmediğini görerek $y = x + 5$; yerine $y = 17$; yerleştirebilir. Tabi burada x volatile yapılrsa derleyici böyle bir optimizasyonu yapmaz.

Ortak Blok (Basic Block)

Bu tema kodu küçültmek için uygulanmaktadır. Eğer kod içerisinde tekrarlanan bloklar varsa bu bloklara tek yerden girilip tek yerden çıkışlıyorsa derleyici bu ortak kodlardan bir tane yerleştirip oraya jump işlemi yapabilir. Örneğin:

```
if (test1) {  
    ifade1;  
    ifade2;  
    ifade3;  
}  
  
*****  
*****/  
  
if (test2) {  
    ifadel;  
    ifade2;  
    ifade3;  
}
```

Burada if deyiminin doğruysa kısmı ortak bloktur. Dolayısıyla tıpkı bir fonksiyon gibi ele alınabilir.

```
if (test1)  
    call REPEAT;  
*****/  
if (test2)  
    call REPEAT;  
*****/  
REPEAT:  
    ifadel;  
    ifade2;  
    ifade3;  
    ret;
```

Göstericilerin Eliminasyonu

Göstericilerin gösterdiği yerlere erişim, yani `*` operatörünün kendisi pek çok sistemde en az iki makine komutuna yol açmaktadır. Örneğin 32 bit Intel mimarisinde `*p = val` gibi bir işlem

```
mov eax, p  
mov [eax], val
```

gibi bir makine koduyla gerçekleştirilir. Bir döngü içerisinde sürekli `*p` ifadesinin kullanıldığını düşünelim.

```
for (;;) {  
    /* */  
    a[i + 2] = *p;  
}
```

Eğer derleyici `p`'nin gösterdiği yerdeki bilginin değişmeyeceğinden emin olsa `*p`'yi bir geçici değişkene ya da yazmaca alır ve onu kullanabilir. Fakat buna emin olabilir mi? Optimizasyon işlemi tek threadli sistemde çalışıldığı varsayılarak yapılmaktadır. Dolayısıyla başka bir threadin `*p`'yi değiştirmesi gibi bir analizi derleyici yapmak zorunda değildir (Programcı `*p` global bir nesneyse göstericiyi volatile yaparak threadler arası kullanımında bu optimizasyonu engellemek isteyebilir.). Derleyici `*p`'nin gösterdiği yerin değişmeyeceğinden emin olmalıdır.

Göstericilerin Eliminasyonu (devam)

Örneğin p göstericisi bir nesneyi gösteriyor olsun o nesnede döngü içerisinde değişiyor olsun. Bu durumda derleyici bu optimizasyonu yapamaz. Örneğin:

```
for (;;) {
    x = i;
    ****/
    a[i + 2] = *p;
}
```

Burada örneğin eğer p x'i gösteriyorsa, derleyici *p'nin değerini yazmaya yerleştirecek hep o yazmacı kullanamaz. İşte derleyicinin buna emin olması gereklidir. Ya da örneğin:

```
for (;;) {
    /**
    a[i + x] = *p;
    func();
}
```

Ya p global değişkeni gösteriyorsa ve func fonksiyonu da onu değiştiriyorrsa? Derleyicinin p'nin global bir nesneyi gösterdiğini anlayabildiğini düşünelim. Eğer func başka bir modülde tanımlanmışsa, derleyici onun kodunu göremeyeceği için artık *p işlemini optimize edemez. Eğer func aynı kaynak dosya içerisinde ise derleyici inceleyerek buna karar verebilir. Derleyicilerin bu analizi yapması vakit alıcı bir işlemidir.

Inline Fonksiyon Açımı

Bir fonksiyon inline anahtar sözcüğüyle bildirilirse derleyici bu fonksiyon çağrıldığında fonksiyon çağrıları değil fonksiyon iç kodunu koda yerleştirir. Örneğin:

```
inline int Kare(int a)
{
    return a * a;
}
```

Şimdi biz bu fonksiyonu şöyle çağrılmış olalım:

```
a = Kare(b);
```

Derleyici muhtemelen şöyle bir kod açacaktır:

```
a = b * b;
```

Gördüğü gibi inline fonksiyonlar makrolara oldukça benzemektedir. Fakat çok daha kolay ve güvenli yazılmaktadır. Ancak inline anahtar sözcüğü bir emir değil tavsiye niteliğindedir. Derleyiciler uzun kodları ya da çok zaman alan döngüler içeren kodları bir fayda sağlanamayacağı gerekçesiyle açmamayırlar.

Döngü Değişmezleri (Loop Invariants)

Döngü değişmezlerinin döngünün dışına çıkartılması en klasik döngü optimizasyonlarından biridir. Örneğin:

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

Döngü içerisindeki fonksiyon çağrılarının döngü değişmezi olarak değerlendirilmesi çoğu kez mümkün değildir. Örneğin:

```
for (;;) {  
    x = Func();  
    /***/  
}
```

Çünkü fonksiyon başka yan etkilere yol açabilir. Örneğin bir global değişkeni değiştiriyor veya bir dosyaya bir şeyler yazıyor olabilir. Programcı bu yan etkinin birden fazla kez oluşmasını sağlamak için çağrıyı döngü içerisinde almış olabilir. Tabi derleyici fonksiyonu tanımlamasını görürse ve bunun bir yan etkiye yol açmadığını belirlerse bu durumda çağrıyı döngü dışına çıkartabilir.

Döngü Değişmezleri (Loop Invariants) (devam)

Örneğin bir karakter dizisinin aşağıdaki gibi dolaşılması kötü bir fikirdir:

```
for (i = 0; i < strlen(s); i++) {  
    *****  
}
```

Derleyiciler standart C fonksiyonlarının ne yaptığını bilmek zorunda değildir. Hatta derleyiciler standart C fonksiyonlarının farkında da değildirler. Dolayısıyla derleyici strlen fonksiyonunun bir yan etkiye sahip olabileceğini düşündüğü için her defasında bunu yeniden çağrılmak zorunda kalır. O halde bu kodun programcı tarafından şöyle düzeltilmesi gereklidir:

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

Fakat pek çok iddialı derleyicide “intrinsic function” denilen fonksiyon kavramı da vardır. Örneğin Microsoft C derleyicilerinde pek çok standart C fonksiyonu “intrinsic function”dır. Bu tür fonksiyonların derleyici ne yaptığını bilir. Hatta bunlar için prototip de istemez. Dolayısıyla bu tür fonksiyonlar döngü değişmezi olarak değerlendirilebilir. Başka ilave ek optimizasyonlara yol açabilir. Örneğin:

```
x = strlen(s);  
y = strlen(s + 1);
```

Burada eğer strlen “intrinsic function” ise derleyici bu fonksiyonun yazı uzunluğunu bulduğunu bilir dolayısıyla ikinci çağrıyı hiç yapmadan $y = x - 1$ gibi bir kod üretебilir.

Döngü Açımı (Loop Unrolling)

```
for (i = 0; i < 10; i++)
    ifade;

for (i = 0; i < 5; i++) {
    ifade;
    ifade;
}
```



Aşağıdaki kod daha fazla yer kaplamasına karşın daha hızlı çalışır. Hatta abartılı olarak tamamen döngüyü kaldırıp peşi sıra 10 tane ifadeyi yazabilirdik. Tabi bunun bir sınırı vardır. Yani milyon kez dönen bir döngünün tam açılması düşünülemez. Fakat 4 kat ya da 8 kat açılım yapılabilir.

Bazen döngü açımları daha karmaşık bir biçimde yapılabilir. Buna “loop unwinding” denir.

Örneğin:

```
for (i = 0; i < 100; i++)
    Func(i);
```

Yerine aşağıdaki kod yazılabılır:

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

Döngü Ayırması (Loop Splitting)

Bazen bir döngünün içerisinde if gibi deyimler varsa bu if deyiminden kurtulmak için derleyici döngüleri ayıralabilir. Örneğin:

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

Yerine aşağıdaki açılım yapılabilir:

```
for (i = 0; i < 100; i += 2)
    ifade1;
for (i = 1; i < 100; i += 2)
    ifade2;
```

Tabi döngüyü bölmek gerçekten bir avantaj sağlıyorsa yapılabilir.

Kod Üretime Aşamasında Yapılan Optimizasyonlar

- Klasik optimizasyonlar ara kod üzerinde ya da pars tree üzerinde yapılmaktadır. Bu optimizasyonlardan geçildikten sonra optimize edilmiş olan kod için üretim yapılır. İşte bazı optimizasyonlar kod üretimine yönelikir. Bunların en önemlileri şunlardır.
 - Komut seçimi (instruction selection)
- Tipik olarak CISC tabanlı işlemcilerde komutların cycle süreleri birbirinden farklı olma eğilimindedir. Yani çok sayıda komut vardır ve bunların çalışma süreleri farklıdır. Halbuki RISC tabanlı işlemcilerde komutların çalışma süreleri eşit olma eğilimindedir. Bu durumda girişi yapabilmek için farklı makine komutları seçilebilir. İşte derleyicinin hız optimizasyonunda en hızlı bir biçimde çalışacak komutları seçmesi gereklidir. Benzer biçimde büyülü optimizasyonunda da derleyici en az yer kaplayacak makine komutlarıyla programcının 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 gerçekleştirilebilir. Çünkü RISC işlemcilerinde komut sayısı azdır ve aralarında ciddi bir performans farkı yoktur.

Yazmaç Tahsisatı (Register Allocation)

- Derleyiciler prensip olarak mümkün olduğu kadar çok değişkeni, mümkün olduğu kadar uzun süre yazmaçlarda tutmak isterler. Genel olarak RISC tabanlı işlemcilerde fazla sayıda yazmaç bulunma eğilimindedir. ALFA gibi çoğu RISC işlemcisinde 32 tane genel amaçlı yazmaç, 32 tane de gerçek sayı yazmacı bulunmaktadır. Fakat Intel, Pentium gibi CISC işlemcilerinde yazmaç sayısı çok azdır. İşte derleyici yazmaç sayısı n olmak üzere ne zaman ve hangi n tane değişkeni yazmaçta tutacağına karar vermelidir. Bu NP tarzda bir problemdir ve çeşitli sezgisel (iyi çözümü veren) çözümleri vardır.

Komut Çizelgelemesi (Instruction Scheduling)

- Pek çok programda bazı ifadelerin ya da deyimlerin yerlerinin değiştirilmesi eşdeğerliği bozmadır. Örneğin:

$x = a;$

$y = b;$

$z = c;$

$k = a;$

- Burada biz $x = a$ ile $k = a$ ‘yı peş peşe getirsek de değişen bir şey olmayacağıdır. Fakat bu sıralama performansı etkileyebilir. İşte komut çizelgelemesi komutların yerlerini değiştirerek bundan bir çıkar sağlamaya yönelik işlemlerdir. Bazı komutların bazı komutlardan önce ya da sonra gelmesi çeşitli nedenlerden dolayı performansı etkileyebilir.

ÖZETLE DERLEME

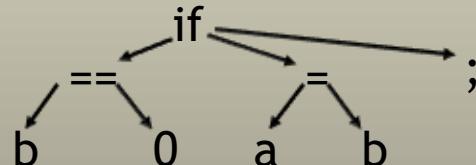
Kaynak kod
(karakter akışı)

if (b == 0) a = b;

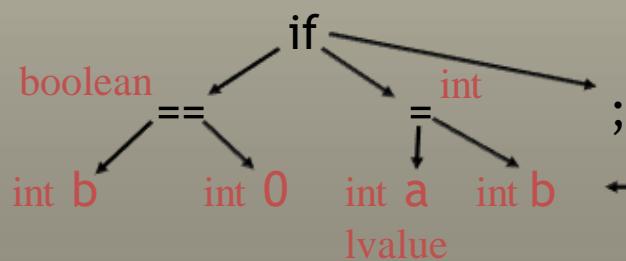
Token akışı

if | (| b | == | 0 |) | a | = | b | ;

Soyut sentaks ağacı
(Abstract Syntax Tree
(AST))



Donatılmış AST

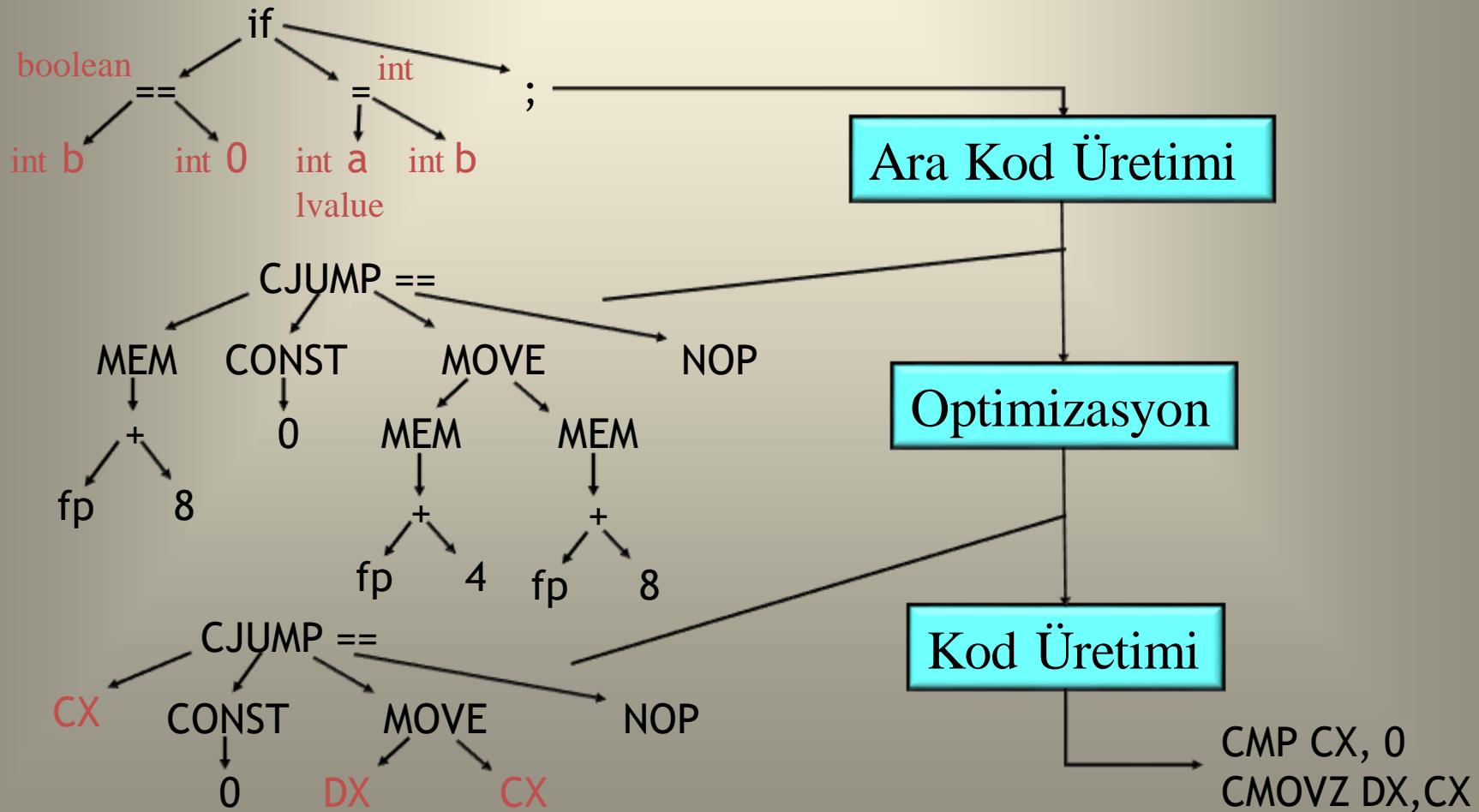


Lexical Analiz

Ayrıştırma

Semantik Analiz

ÖZETLE DERLEME



Örnek Derleme

Kaynak Program

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Ara Kod Üreticisi

Ara Kod

Kod Optimizasyonu

Hedef Dil Üreticisi

Hedef Dil Programı

Kaynak Kod:

```
cur_time = start_time + cycles * 60
```

Örnek Derleme

Kaynak Program

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Ara Kod Üreticisi

Ara Kod

Kod Optimizasyonu

Hedef Dil Üreticisi

Hedef Dil Programı

Kaynak Kod :

`cur_time = start_time + cycles * 60`

Lexical Analiz:

`ID(1) ASSIGN ID(2) ADD ID(3) MULT INT(60)`

Örnek Derleme

Kaynak Program

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Ara Kod Üreticisi

Ara Kod

Kod Optimizasyonu

Hedef Dil Üreticisi

Hedef Dil Programı

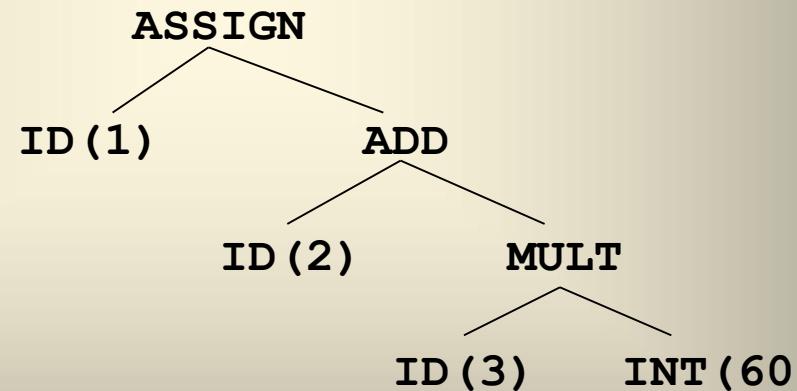
Source Code:

```
cur_time = start_time + cycles * 60
```

Lexical Analiz:

ID (1) ASSIGN ID (2) ADD ID (3) MULT INT (60)

Syntax Analiz:



Kaynak Program

Örnek Derleme

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Ara Kod Üreticisi

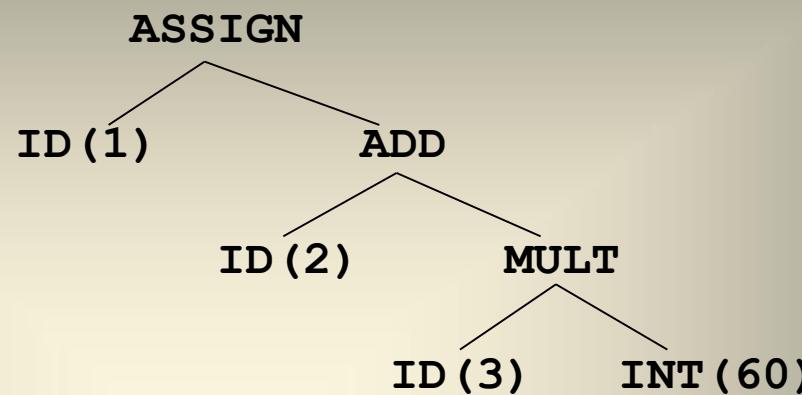
Ara Kod

Kod Optimizasyonu

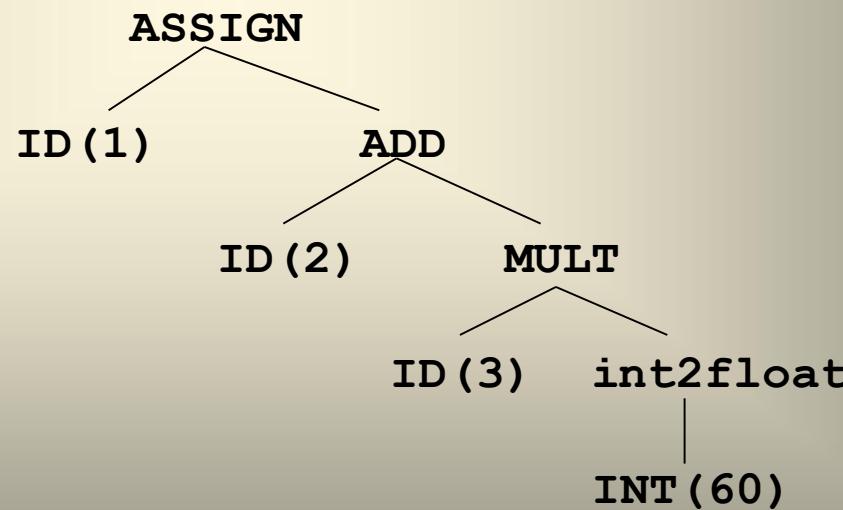
Hedef Dil Üreticisi

Hedef Dil Programı

Syntax Analiz:



Semantic Analiz:



Kaynak Program

Örnek Derleme

Lexical Analyzer

Syntax Analyzer

Semantic Analyzer

Ara Kod Üreticisi

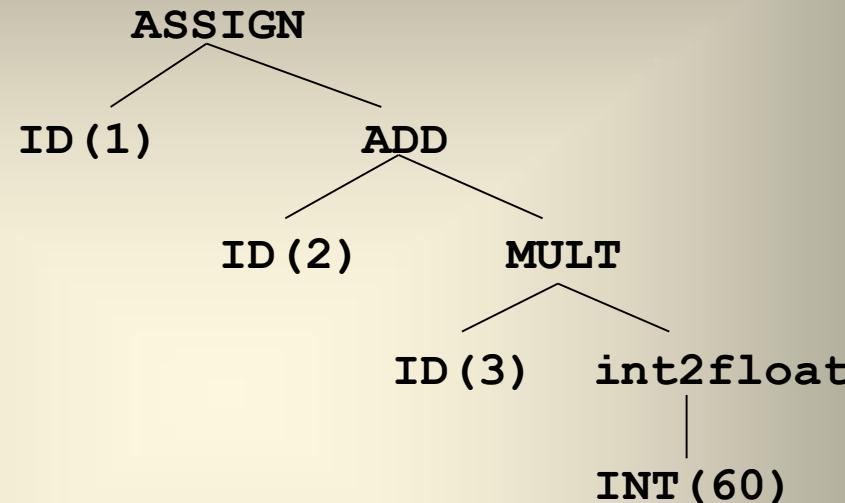
Ara Kod

Kod Optimizasyonu

Hedef Dil Üreticisi

Hedef Dil Programı

Semantic Analiz:



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Örnek Derleme



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimize Kod (step 0):

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Örnek Derleme



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimize Kod (step 1):

```
temp1 = 60.0
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Örnek Derleme



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimize Kod (step 2):

```
temp2 = id3 * 60.0
temp3 = id2 + temp2
id1 = temp3
```

Örnek Derleme



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimize Kod:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

Örnek Derleme



Ara Kod:

```
temp1 = int2float(60)
temp2 = id3 * temp1
temp3 = id2 + temp2
id1 = temp3
```

Optimize Kod:

```
temp1 = id3 * 60.0
id1 = id2 + temp1
```

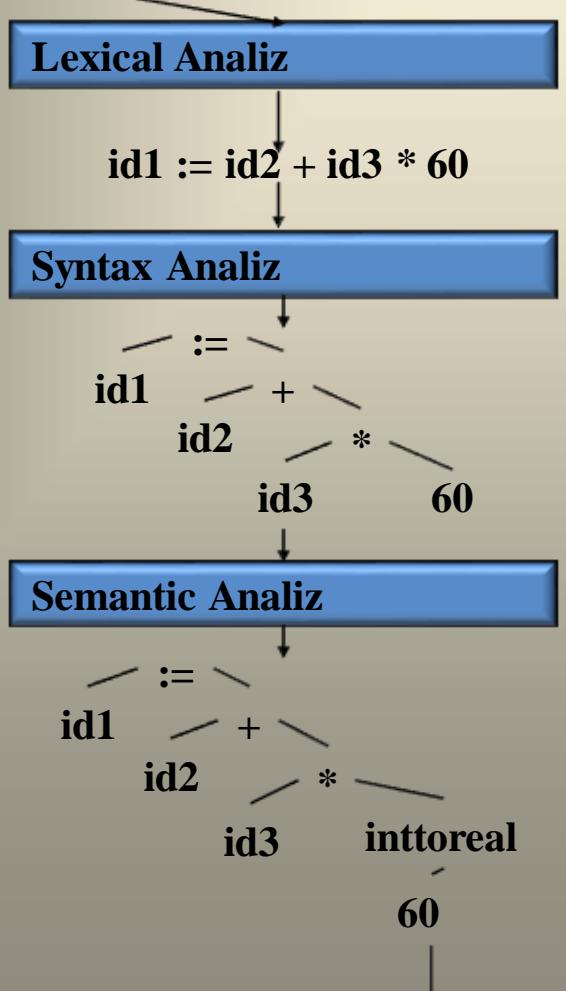
Hedef Kod:

```
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1
```

Hedef Dil Programı

Derleme ÖZET

```
cur_time = start_time + cycles * 60
```



```

graph TD
    A[Ara Kod Üreteci] --> B[temp1 := inttoreal(60)  
temp2 := id3 * temp1  
temp3 := id2 + temp2  
id1 := temp3]
    B --> C[Kod Optimizasyonu]
    C --> D[temp1 := id3 * 60.0  
id1 := id2 + temp1]
    D --> E[Kod Üreteci]
  
```

The diagram illustrates the compilation process. It starts with the **Ara Kod Üreteci** (Intermediate Code Generator) phase, which processes the input code to produce initial assembly-like intermediate code. This is followed by the **Kod Optimizasyonu** (Code Optimization) phase, which refines the intermediate code. Finally, the **Kod Üreteci** (Code Generator) phase produces the final optimized assembly code.

```
MOVF id3, R2  
MULF #60.0, R2  
MOVF id2, R1  
ADDF R2, R1  
MOVF R1, id1
```

Dil Çevrim Yöntemlerinin Karşılaştırılması

- Yorumlama ve derleme yöntemleri, bir programlama dilinin makine diline çevrilmesi sürecinde, çeşitli kriterler açısından üstünlükler veya zayıflıklar içerirler.
- İlerleyen sayfalarda iki yöntem, çevrim sürecinde bilgisayar kaynaklarının etkin kullanımı ve kullanıcıya hata bildirme yöntemleri açısından karşılaştırılmıştır.

Zaman Etkinliği Açısından

- Yorumlama yaklaşımı ile dil çevriminde bir deyimin çalıştırılması için gerekli makine diline çevrim süreci, deyimin her çalıştırılmasında aynen yinelenmelidir.
- Diğer taraftan derleme yaklaşımında, programdaki her deyim için gerekli makine kodu, deyim kaç kez çalıştırılırsa çalıştırılsın, sadece bir defa oluşturulur. Yani bir programın makine koduna çevrimi tamamlandıktan sonra oluşan makine kodu, program kaç kez çalıştırılırsa çalıştırılsın, program değiştirilmediği sürece aynen kullanılabilir. Bu nedenle derleme yöntemi, yorumlamaya göre zamandan kazanç sağlar.

	Zaman
Derleme	✓
Yorumlama	✗

Bellek Kullanma Etkinliği Açısından

- Derleme yaklaşımı ile dil çevriminde her yüksek düzeyli dil deyimi, onlarca makine dili komutuna dönüştürilebilirken, yorumlamada yüksek düzeyli dil deyimleri, orijinal şekillerinde kalırlar ve onların çalıştırılması için gerekli komutlar da yorumlayıcının altprogramları olarak bulunur.
- Dolayısıyla yorumlama, derleme yaklaşımına göre daha az bellek kullanabilir. Öte yandan, çalışma sırasında yorumlayıcının tüm altprogramları bellekte tutuluyorsa, dilin sadece az sayıda deyimini kullanan programlar için yorumlayıcı bellek israfına neden olabilir.

Bellek Kullanma Etkinliği Açısından

	Zaman	Bellek Kullanma
Derleme	✓	✗
Yorumlama	✗	✓

Çalışma sırasında yorumlayıcının tüm altprogramları bellekte tutuluyorsa, dilin sadece az sayıda deyimini kullanan programlar için yorumlayıcı bellek israfına neden olabilir.

Hata Bildirme Yöntemleri Açısından

- Derleyiciler ve yorumlayıcılar, çalışma sırasında hataların kullanıcıya bildirilmesi açısından farklılık gösterirler.
- Yorumlayıcılar her komutu birer birer ele aldıkları için, hataların kullanıcılarına bildirilmesi doğrudan gerçekleşir.
- Derleme yönteminde ise hatalar, tüm program deyimlerinin makine koduna çevrilmesi bittikten sonra toplu olarak bildirildiği için, hata kaynağı deyimlerin belirlenmesi, yorumlama yöntemine göre güçtür.

Hata Bildirme Yöntemleri Açısından

	Zaman	Bellek Kullanma	Hata Bildirme
Derleme	✓	✗	✗
Yorumlama	✗	✓	✓

Taşınabilir Kod

- Dil çevriminde bütünüyle yorumlama ve bütünüyle derleme yöntemleri, iki uç durumdur. Bazı programlama dilleri, iki yöntemin birleştirilmesi ile gerçekleştirirler.
- Bir program, kaynak program üzerinde basit düzenlemeler yapılarak bir sanal makinenin daha sonra yorumlanacak olan makine kodu olarak nitelenebilen bir ara koda çevrilebilir. Bu çözüm, ağırlıklı olarak derlemeye dayanır ve farklı makinelerde çalıştırılabilen **taşınabilir kod** üretmek amacıyla kullanılabilir.

Taşınabilir Kod



- Java dili bu tür dil çevrimini uygulamaktadır. Java programları, Java *bytecode*'u adı verilen bir ara koda dönüştürüldükten sonra yorumlanır.

Diğer Çeviriciler

- Derleyici derleyiciler (*compiler compilers*): Herhangi bir programlama dili için geliştirilmiş olan ve o dil için derleyici yaratan programlardır.
- Çapraz bütünlükleştirici (*cross assemblers*): Bir bütünlükleştirici dilinden diğer bir bütünlükleştirici diline çevrim işlemini gerçekleştiren programlardır.
- Çapraz derleyiciler (*cross compilers*): Herhangi bir programlama dilinde yazılmış olan bir program diğer bir programlama diline çevrilebilmektedir.

0.5. Programlama Dillerinin Yazılım Yaşam Döngüsündeki Yeri

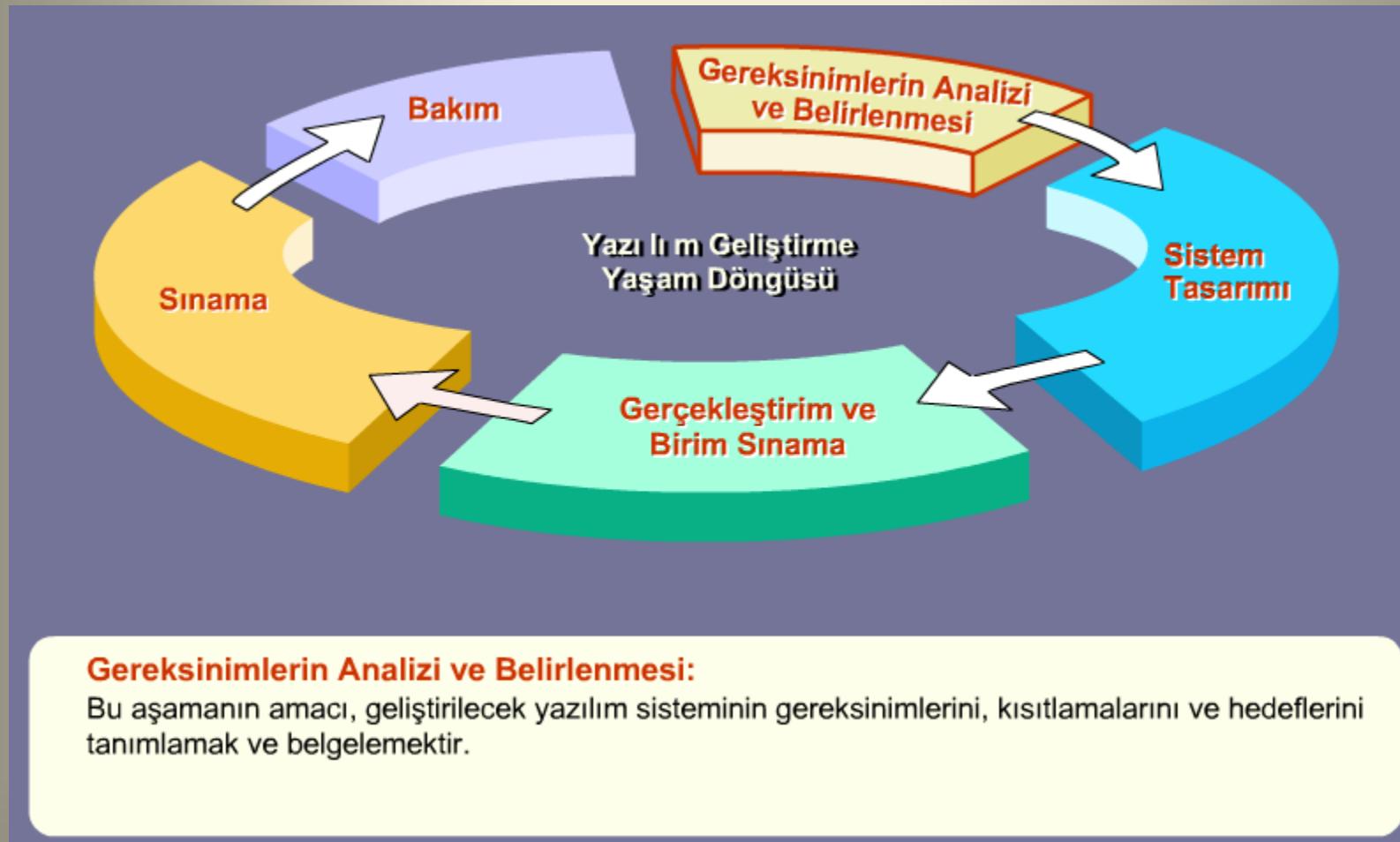
- Önceki bölümlerde genel olarak tanıtılan programlama dilleri, yazılım mühendisliğinin temelini oluşturan yazılım geliştirme yaşam döngüsünün gerçekleştirim aşamasında kullanılır.
- Bu bölümde, programlama dillerinin yazılım yaşam döngüsündeki yeri.

Yazılım Geliştirme Yaşam Döngüsü Nedir?

- Yazılım geliştirme yaşam döngüsü temel olarak 5 aşamadan oluşmaktadır.



Geliştirme Yaşam Döngüsü Nedir?



Gereksinimlerin Analizi ve Belirlenmesi:

Bu aşamanın amacı, geliştirilecek yazılım sisteminin gereksinimlerini, kısıtlamalarını ve hedeflerini tanımlamak ve belgelemektir.

Yazılım Geliştirme Yaşam Döngüsü Nedir?



Sistem Tasarımı:

Sistem tasarımı aşaması, gereksinimleri, donanım ve yazılım sistemlerine ayırır. Bunun sonucu olarak sistemin mimarisi tanımlanır ve yazılımin, belirtilen gereksinimleri karşılayacak şekilde modüllere ayırımı ile devam edilir. Bu aşamada izlenen tasarım yöntemi, sonuctaki ürünün kalitesini, özellikle anlaşılabilirlik ve değiştirilebilirliğini önemli ölçüde etkiler.

Yazılım Geliştirme Yaşam Döngüsü Nedir?



Yazılım Geliştirme Yaşam Döngüsü Nedir?



Yazılım Geliştirme Yaşam Döngüsü Nedir?



Bakım:

Sistemin kullanılması sırasında fark edilen hataların düzeltilmesi veya sistemin kullanımı sırasında ortaya çıkan gereksinimlerin eklenmesi için, sistemde değişiklikler yapılması gerekli olabilir. Bu değişiklikler, yazılım geliştirme yaşam döngüsünün en uzun süren aşaması olan bakım aşamasını oluşturur.

ÖDEV

- Lex & Yacc , Flex & Bison konularını araştırıp konuya ilişkin dokümanları, Word ve Powerpoint belgesi halinde hazırlayınız.
 - Örnek program kodları
 - Kaynak dokümanlar

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