Programlama Dillerinin Prensipleri

Mainde string args dışarıdan parametre almak için kullanılır.

https://www.youtube.com/watch?v=PDzWHsBE4-w

C'de argv[0] cmd üzerinden programın adını yazarak başlattığımız için programın adıdır.

Javada ilk hangi parametre alınırsa argv[0] odur.

C'de setlocale() double ve float değişkenlerde problem yaratmaktadır. Amerikan standartı ".", Türkiye "," olduğundan dolayı.

C'de class yoktur.

C'de String yoktur, charlarla ifade edilir. `char *a` şeklinde string alınabilir. (karakterler dizisi)

`printf("Merhaba Dunya");` şeklinde yazdırma yapılır.

`scanf("%d", &x);` x int veri tipindeki değere alınacak değeri atar.

%d → int

%f → float

%lf → long float yani double anlamına gelir

%c → char

%s → string

%x → adres yazdırma (yani hexadecimal)

Programın çalışıp kapanmaması için main'e `return 0;` öncesi `getchar();` eklenebilir.

Programlama Dilleri Seviyeleri

Makine Dili

Alçak Seviyeli PD → Assembly

Orta Seviyeli PD → C, Ada

Yüksek Seviyeli PD → Fortran, Python

Çok Yüksek Seviyeli PD → C#, Java, Oracle

Programlama Dillerinin Uygulama Alanlarına Göre Sınıflandırılması

Bilimsel ve Mühendislik Uygulama Dilleri → Pascal, C, Fortran

Veritabanı Dilleri → MsSQL, Oracle Forms, XBASE

Genel Amaçlı Programlama Dilleri → Pascal, C, Basic, Java, Python

Yapay Zeka Dilleri → Prolog, Lisp

Modelleme Yapmak için Simülasyon Dilleri → GPSS, Simula67(İlk nesne yönelimli dil)

Makro Diller(Script) → awk, Perl, Python, Tcl, JavaScript

Sistem Programlama Dilleri → C

Ticari Uygulamalara Yönelik Programlama Dilleri → Cobol

Nesne yönelimli dilin temeli Algol 60'dır.

4 farklı tasarım paradigması vardır. Bunlar ve örnekleri,

C, Fortran, Pascal, Cobol → Emir Esaslı

Java, C# → Nesne Yönelimli (Olayı 1-Kapsülleme 2-Kalıtım 3-Çok Biçimlilik)

Lisp → Fonksiyonel Programlama

Prolog → Mantıksal Programlama

C++ → Emir Esaslı ve Nesne Yönelimli

Yorumlayıcı Dil → Kodun satır satır makine diline çevrilmesi(Python)

Derleyici Dil → Kodun bir bütün olarak okunup makine diline çevrilmesi

Java bellekteki çöpleri kendisi temizler.

Java taşınabilirliği JDK sanal makinesi ile yapar. Kodlar Java sanal makinesine(JRE) yazılmış olur(byte), o da makine koduna çevirir. Bundan dolayı taşınabilir olur. C# ise bunu CLR ile yapar.(.NET Core)

Script Dilleri → İstemci ve Sunucu taraflı olarak 2'ye ayrılır. JavaScript istemci taraflı bir dildir yani kod elde bulunmak zorundadır, yorumlayıcı dildir. Yani kod erişime açık olur çünkü yorumlamalı olduğundan dolayı olmak zorunda. Eğer kodun erişinilmemesi isteniyorsa sunucu taraflıya geçilmesi gerekir. Sunucu taraflıda makine koduna dönüşüm işlemi sunucuda yapılır. İstemcide ise dönüşüm bilgisayarda yapılır. PHP sunucu tabanlıya örnektir.

C'de true false yoktur, 0 false, 0 hariç diğer sayılar true'dur. Örneğin if(100) c için true iken javada hata verir.

C'de sizeof, değişkenin bellekteki boyutunu gösterir yani içindeki değer ne olursa olsun default olarak kapladığı alanı söyler → `printf("%d", sizeof(int));` (sizeof int döndürür kaç byte %d)

Bir program içerisinde ya float ya double kullan her seferinde! Yani double ve float birlikte kullanılmamalıdır.

**Float ve double == ifadesinde aynı değere sahip olsalar da sonuç false çıkar. Çünkü double float'tan daha fazla yer kaplar, virgülden sonra daha fazla rakam barındırır. (Hem Java hem C)

C'de örneğin `double pi=3.14;` atanmış ve `int a=pi;` denilirse veri kaybı oluşur, yani C otomatik olarak tip dönüşümü yaparak int haline getirir ve pi 3 olur.

Javada `int a=pi;` denmesine izin vermez çünkü güçlü tip kontrolü vardır. Eğer veri kaybını gerçekten istiyorsak yani bu olayın farkındayım demek istiyorsak kendimiz tip dönüşümünü

`int a=int(pi);` şeklinde yapmalıyız.

C'de const yani sabit tanımlaması yapılıyorsa tanımlandığı yerde muhakkak atama yapılmalıdır, sonradan atama yapılmasına izin vermez.

İzin vermez;

```
const double yercekim; yercekim=10;
```

İzin verir;

```
const double yercekim=10;
```

Aynı zamanda C'de main üzerine `#define degiskenAdi` şeklinde de sabit tanımlaması yapilabilir.

Javada ise sabit tanımalaması yapıp ataması daha sonradan C'de izin verilmediği şekilde yapılabiliyor. Javada sabit tanımlaması const yerine final anahtar kelimesiyle yapılır.

Javada var değişken tipi bulunur ve bu var ilk hangi tipte veri alırsa o tipe dönüşür. Örneğin bir dizi yapısı bulunuyorsa ve bu dizi ilk olarak int değer alırsa var olarak tanımlanmış o dizi int tipinde bir diziye dönüşmüş olur.

https://www.youtube.com/watch?v=l8Z6JJAaXUs

Dillerin Çevrimi (Derlenme)

1. kaynak kod → c, c++ vs.
2. lexical analiz
3. syntax analiz
4. semantic analiz

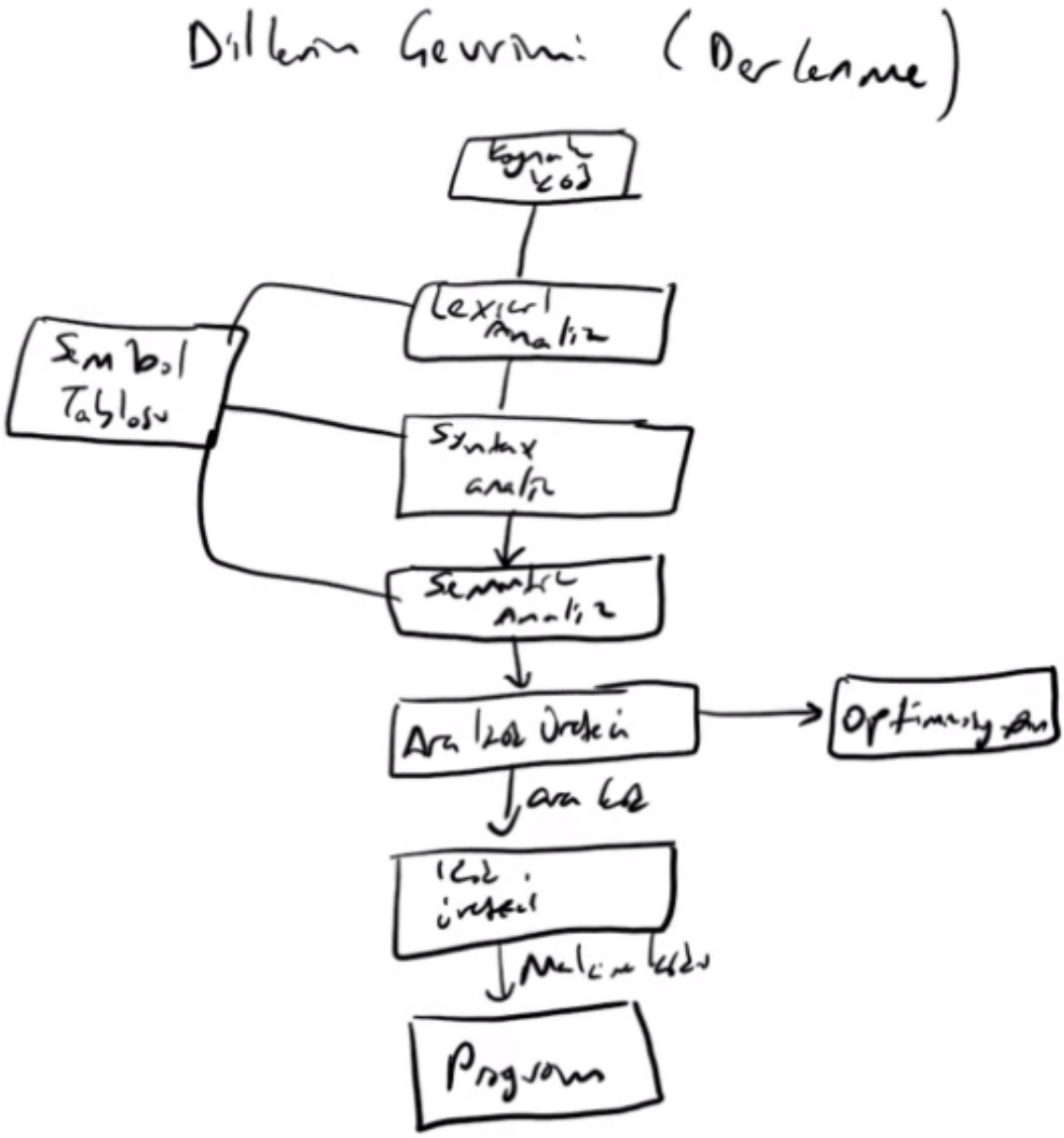
(analiz biçimleri sembol tablosundan referans alıyor)

5. semantic analizi geçmiş olan kod artık derlenme esnasında hata vermeyeceğinden emin olunan yani makine koduna dönüştürülmeye hazır olan kod manasına gelip sonrasında ara kod üretici denilen kısma girer. Bu kısıma kadarki kısma ön uç(makine bağımsız), bundan sonraki kısma arka uç(makine bağımlı) denir.

- Her derleyicide zorunlu olmasa da opsiyonel olarak bulunan optimasyon aşaması da bulunur, varsa ara kod üreticiden sonra kod üretici arasında yapılır.

6. Kod üretici → Makine kodunu üretir

7. Program → exe, çalıştığı anda üretilen makine kodu önbelleğe atılır ve program çalışmaya başlar.

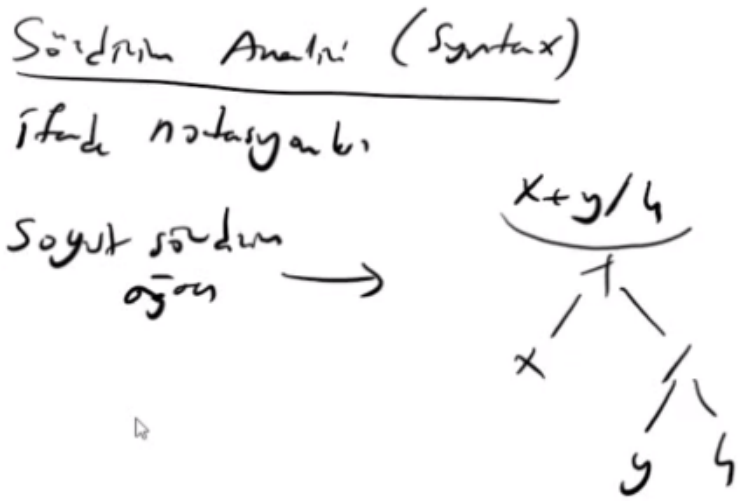


Lexical analiz (sözcüksel analiz) → Kaynak kodu en küçük anlamlı birimlere(tokenlara) parçalar ve bunları niteleyerek gruplar haline getirir. Yani örneğin int a=10 olarak yazılmış kodu `int` , `a` , `=` , `10` olarak ayırmış olur. Her sözcük, ifade bir tokeni ifade eder. Aynı zamanda gruplamak, nitelemek için de int→ tür, a→ tanımlayıcı, =→ atama, 10→ tamsayı şeklinde niteler.

Syntax analiz (sözdizim analiz) → Kaynak kodun dilin gramerine uyup uymadığını kontrol eder. Gramerlerin birden fazla gösterim şekilleri vardır. Örneğin CFG, BNF, E-BNF.

CFG'de Başlangıç NonTerminali-Terminaller-NonTerminali-Sabitler-Kurallar vardır. Terminal durak anlamına gelir.

Analizde öncelikle ifade notasyonlarının soyut sözdizim ağacı oluşturulur. Bir ifadenin notasyonu 1'den fazla sözdizim ağacı oluşturuyorsa orada belirsizlik vardır yani gramer hatası vardır.



BNF kuralları, Örn: Reel sayılar için;

: : = sembolünün okunuşu "olabilir" ve | sembolünün okunuşu "veya" gibidir. Daha sonra,

---

<reel sayı>	: : =	<tam sayı - kısım> . <kesir>
<tam sayı - kısım>	: : =	<sayı> | <tam sayı - kısım> <sayı>
<kesir>	: : =	<sayı> | <sayı> <kesir>
<rakam>	: : =	0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<kesir>	: : =	<rakam> | <rakam> <kesir>

Reel sayılar için BNF kuralları

şeklinde tanımlanırlar. Sol taraf nonterminal, sağ taraf terminaller veya nonterminaller. Nonterminal ifade sağ tarafta da bulunuyorsa yine nonterminaldir, terminal değildir.

< > arasında kalan ifadeler NonTerminali ifade eder. Yani örneğin 5. satır <kesir> için <rakam> da bir nonterminaldir. <kesir> içerisinde sağ tarafta yeniden kesir çağırılması recursivedir.

-, +, * veya rakam vs. gibi ifadeler de terminalleri ifade eder. ('|' terminal değildir.)

Karakterler 'c' olarak yani ' arasında gösterilir.

BNF'de `::=` CFG'de `->` ok işareti olarak kullanılır. Ayrıca CFG'de <> ifadeleri kullanılmaz.

Örneğin rakam → 0|1|2|3|4|5|6|7|8|9

Sağ ve Sol Rekürsif Gramer

Örn şöyle bir gramer olsa elimizde, (sol rekürsif)

S→Ra|a

R→ab|Rb

Şöyle ilerlersek, (yerlerine koyarak)

S→Ra→Rba→Rbba→abbba

görüldüğü üzere sürekli sol taraftan genişletilmiş. Yani bu gramere sol rekürsif gramer denir.

Başka bir örnek gramer, (sag rekürsif)

S→aR|a

R→ab|bR

Şöyle ilerlersek, (yerlerine koyarak)

S→aR→abR→abbR→abbab (CFG açılımı böyle dallandırmalı gösteriliyor)

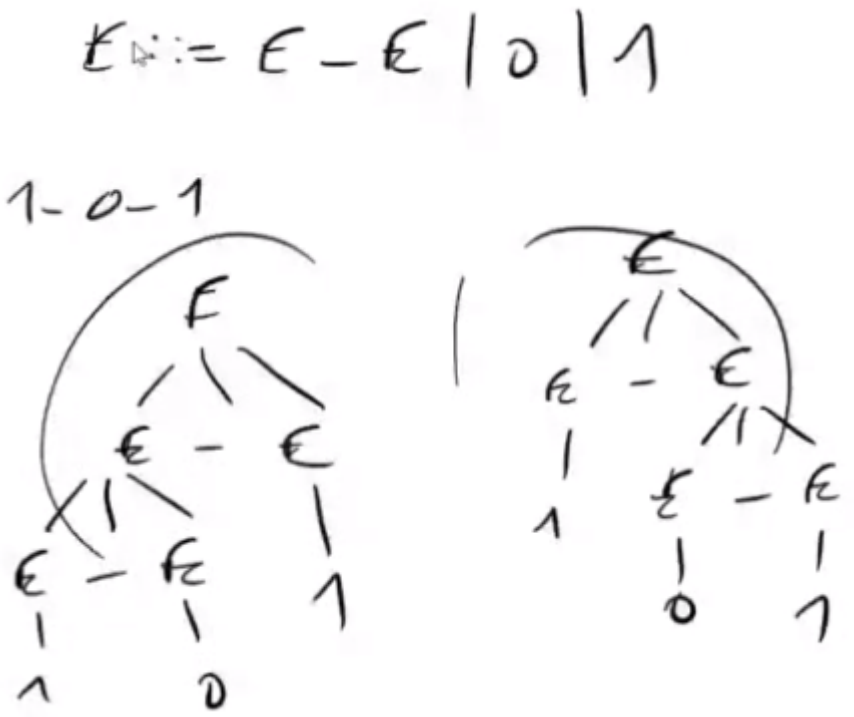
görüldüğü üzere burada da sürekli sag taraftan genişlemiş. Buna da sag rekürsif gramer denir.

İstenilen durum yalnızca biri olmasıdır yani ya sag ya da sol rekürsif olunması. Fakat her ikisinin de olduğu durumda belirsizlik içerme ihtimali çok fazladır yani belirsizlik vardır denir. Böyle bir durumda da bu grameri ya sağ ya da sol rekürsife çevirmek gerekir.

Örn;

<E> ::= <E-E>|0|1 (Burada E nonterminal, -, 0, 1 terminaldir.

Diyelim ki bu gramere göre 1-0-1 üretilebiliyor mu diye soruldu. Bu durumda nonterminal genişletilir. Fakat görüleceği üzere hem sağdan hem de soldan genişletilebiliyor ve her ikisi de aynı sonucu verebiliyor. Bu gramer belirsizdir, bu belirsizliği çözmek için de yeni bir non terminal eklenmesi gerekir. Bu yeni eklenen non terminalle beraber gramer sağ veya sol rekürsif yapılır. Bu kadarının bilinmesi bu ders kapsamında yeterlidir. (BNF açılımı böyle dallandırmalı gösteriliyor)



E-BNF;

BNF'ye ek olarak Yineleme(→ { }) bulunur. Örneğin ifadeleri bitirmek için kaynak kodda ; kullanırız ve bu ve bunun gibi ifadeler için ifade-listesi tanımladığımızı varsayalım.

<ifade-listesi> ::= £ | <ifade>;<ifade-listesi>(BNF gösterimi. £ NULL ifade demek)

<ifade-listesi> ::= {<ifade>;} (E-BNF gösterimi bu şekilde olur. { } içerisi hiç(NULL) veya çok tekrar edilebilir anlamına gelir.)

BNF'ye ek olarak Seçimlik(→ [ ]) bulunur. Örneğin gerçel sayılar,

<gercelsayi> ::= <tamkisim>.<kesir> | .<kesir> (BNF gösterimi. Örneğin burada tamkisim seçimlik oluyor, kullanıla da billir kullanılmaya da bilir. | sonrası . <kesir>'de örneğin 0.1561 olabilir çünkü. Yani tam kısım ya vardır ya yoktur, en fazla 1.)

<gercelsayi> ::= [<tamkisim>].<kesir> (E-BNF gösterimi bu şekilde olur.)

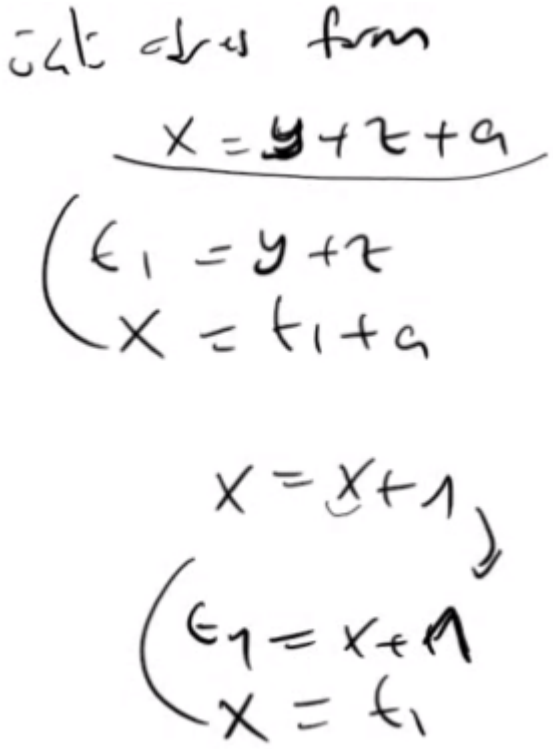
BNF'ye ek olarak Değiştirme(→ *,+) bulunur. * işareti { } yani yinelemeyle aynı işi yapar. +'da en az 1 defa olacak demektir yani 1 veya çok. Örneğin değişken tanımlama olsun,

<id> ::= <karakter> | <id><karakter> | <id><rakam> (BNF gösterimi)

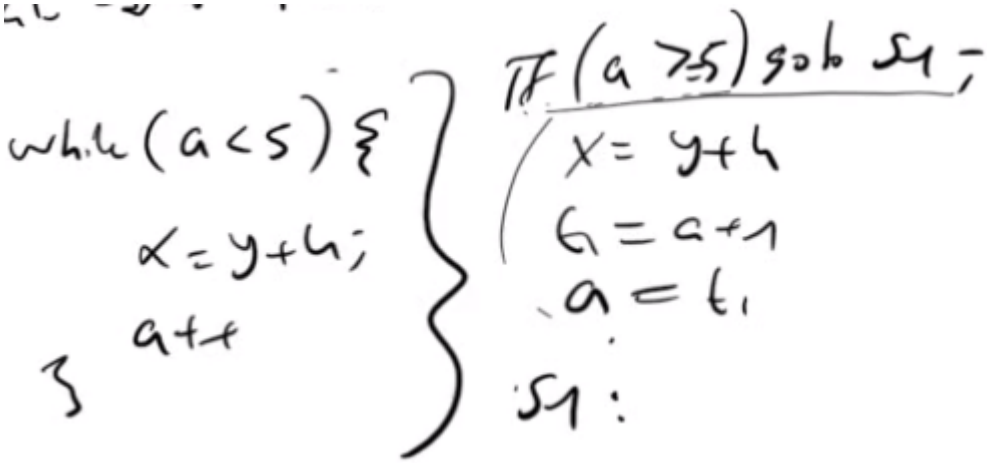
<id> ::= <karakter>(<karakter>|<rakam>)* (E-BNF gösterimi. Ortak paranteze almış gibi olduk. Önce karakter zorunlu, devamında ya karakter ya rakam dedik)

Semantic Analiz (Anlamsal analiz) → Sözdizimsel olarak tanımlanamayan kuralların anlamsal olarak tanımlandığı yerdir. Örneğin Java dilini düşünecek olursak double olan bir değişkeni int'e atmak istersek bunu syntax ve lexical analiz yakalayamaz, burada semantic analiz devreye girer.

Ara Kod Üretici → Makine koduna çevriminin daha kolay yapılabileceği ara bir koda çevrildiği bölümdür ve olması şarttır. Tüm ifadeleri üçlü adres form'a dönüştürür. Örn;



üçlü adres form'da while, forlar bulunmaz. Bunlar yerine if ve goto ifadeleri bulunur çünkü goto ifadelerinin dönüşümü daha kolaydır. Sol taraftaki while S1 olarak ifade edilir.



Varsayalım ki zorunlu olmasına rağmen ara kod üreticisi kısmını kaldırdık; bu durumda semantic analiz sonrası her makine için farklı kodlar yazmak gerekir. Örneğin Java'da bu kısımda JRE vardır.

Optimizasyon → Ara kod üretimi esnasında yapılır. Hiç kullanılmayan bir değişkenin tanımı sonrası bu esnada o değişkenin yok edilmesi bir optimizasyon örneğidir. Veya örneğin;

```
while(....){
  a=5;
  ....
}
```

şeklinde bir döngümüz var fakat bu döngü içerisinde aslında a değişkeni ile hiçbir işimiz yok ve bu döngünün 1000 defa çalıştığını varsayarsak her defasında boş yere a=5 atamasını sürekli yapıyor. Optimizasyon aşamasında bunu fark edip bunu şu şekilde düzenler;

```
a=5;
while(....){
  .....
}
```

Veya ölü kod denilen kodları her zaman çıkartır. Örneğin,

```
x=100;
y=200;
if(x>y){
  ...
}
```

Buradaki if bloğu komple ölü kod olarak geçer çünkü atamalar yapılmış ve x hiçbir zaman y'den büyük olamaz. Optimizasyon esnasında bu blok komple silinir.

Kod Üretici → Derleyiciye göre farklılıklar bulunmaktadır. Örneğin C için direkt makine koduna çevirir. Visual Studio üzerinde kod yazılmışsa C++, C# içinse durum böyle değil çünkü oluşturulan exe'yi çalıştırabilmek için muhakkak .NET Framework gerekmektedir.

Derleyicide yapılan işlemler bu şekilde 1 defa kodun bütününde işleme tabi tutulur ve herhangi bir hata varsa program hiç açılmaz.

****Yorumlayıcı ile farkı ise yorumlayıcı tüm bu işlemleri her satırda tekrarlar.**

Diziler C/C++'da da Java'da da ilk elemanın adresi olarak tutulur.

Stringler de C'de char dizileri olarak ifade edildiğinden aynıdır. Stringde *degiskenAdi(%c) ile dizinin ilk elemanına, degiskenAdi(%s) ile tamamına erişebiliriz.

C'de `int x[3];` şeklinde int tipinde 3 elemanlı x dizisi oluşturulur. `int x[]={10,20,30};` şeklinde de yine aynı şekilde stack bellek bölgesinde elemanları 10,20,30 olan 3 elemanlı x dizisi oluşturmuş oluruz. C'de dizinin boyutunun girilmesi zorunludur. Dizi oluşturulduktan sonra C ve C++ dizi elemanlarını otomatik olarak rastgele saçma sapan sayılar yapar. Yani bu dillerde dizi oluşturduktan sonra muhakkak tüm elemanları önce 0 yapılmalıdır. Javada bunun tersine bizim yerimize 0'lar.

C'de heap bellek bölgesinde dizi oluşturmak içinse `int *dizi=malloc(sizeof(int)*diziBoyutu);` şeklinde dizi oluşturulabilir. Eğer tüm elemanlarının rastgele sayılar olması yerine 0 olmasını istiyorsak bunu calloc ile yapabiliriz fakat zaman açısından calloc daha çok vakitte yer ayırma işlemini tamamlar.

Java'da stackte dizi oluşturmak için `int []x={10,20,30}` şeklinde dizi oluşturulabilir.

Heap bellekte oluşturmak için ise `int []x=new int[5]` → şeklinde 5 elemanlı int tipinde 1 boyutlu x dizisi, `int [][]x=new int[5][3]` şeklinde ise 2 boyutlu x dizisi oluşturulur ve oluşturduğu bu dizinin tüm elemanlarını otomatik olarak "0" yapar.

veya `int []x=new int[] {10,20,30}` şeklinde de heapte dizi oluşturulabilir.

Java'da bir string ifade şu şekilde tekrar ettirilebiliyor,

```
String str="SAU";
String tekrarEdilmis=str.repeat(5);
```

```
System.out.println(tekrarEdilmis);
->SAUSAUSAUSAUSAU
```

Java'da bir string ifade şu şekilde satırlara ayrılabiliyor, yani \n gördükçe ayırıyor

```
String str2="Sakarya Universitesi\nBilgisayar Muhendisligi";
Stream<String> satirlar=str2.lines();
System.out.println(satirlar.findFirst());
->Sakarya Universitesi
```

C'de [] şeklinde bir dizi oluşturmakla string için karakter dizisi oluşturmak aynı şey değildir. Görüleceği üzere 2. fonksiyondan boş çıktı veriyor çünkü fonk içi tanımlanan str değişkeni bir dizi ve return edilirken fonksiyon bittiği için bellekten silinmiş oluyor ve print etmek isteyince adresi boş olduğu için yazdıramıyor. 1. fonksiyonda ise char *str="Sakarya"'nın double x=10.5 gibi bir ifadeden farkı olmadığı için return edilebiliyor.

```
char* SehirDondur(){
  char* str="Sakarya";
  return str;
}
char* SehirDondur2(){
  char str[]="Ankara";
  return str;
}
printf("%s\n",SehirDondur());
printf("%s\n",SehirDondur2());
->Sakarya
->
```

C'de bir yeri gösterme olayı yani adresi tutma işi, * ile yapılır.

```
int x=100,y=50;
int *p=&x;
int *r=&y;
int *tmp=p;
p=r;
r=tmp;
printf("%d\n",*p);
printf("%d\n",*r);
->50
->100
```

Pointer yani * adres tutabilen demektir. *p x'in adresini tutuyor, *r y'nin adresini tutuyor yani p x'in, r de y'nin adresini tutuyor. Bunların refere ettikleri yerleri değiştirebilmek için *tmp tanımlayıp değiştiriyoruz ve en sonunda tam tersi duruma geliyorlar.

Java'da * yoktur fakat referans vardır. Sınıftan nesne üretilmesiyle olur.

```
public class Sayi {
  public int deger;
  public Sayi(int dgr) {
    deger=dgr;
}
Sayi p=new Sayi(100);
Sayi r=new Sayi(50);
Sayi tmp=p;
p=r;
r=tmp;
System.out.println(p.deger);
System.out.println(r.deger);
->50
->100
```

Burada tek 1 referans vardır, o da tmp'dir. p'yi gösterir.

C'de static olarak tanımlanan değişken static bellek bölgesinde program çalıştığı anda oluşur ve program sonlanana kadar silinmez.

```
void f(){
  static int h=10;
}
int main(){
  ....
}
```

Aynı işlem static Java için örneğin yukarıdaki Sayi sınıfında

`public static int deger;`

demiş olsaydık tüm nesnelerin değerinin aynı olmasını sağlamış olurduk. Yani p nesnesi üretip değer 100 gönderilince deger=100 olur, sonrasında r nesnesini üretince 50 gönderince bu defa o sınıfın tüm nesnelerinin degeri 50 olmuş olacaktı. Yani o durumda r.deger de p.deger de 50 sonucunu verirdi. Static olan

üyeye Sayi.deger şeklinde de erişilebilir çünkü statik, ortak üye. Yani static olarak tanımlanan şey her nesne üretildiğinde üretilmiyor, 1 defa üretiliyor ve hep aynı kalıyor. Sınıf içerisinde atama yapılmazsa varsayılan 0 olur. Kısaca static, sınıftan nesne üretmeden de kullanılabilmeyi ve tüm nesneler için aynı değere sahip olmayı sağlıyor.

C'de heap bellekte nesne oluşturmak için malloc, heapten belleğe iade için free kullanılır. Malloc ile yer ayrıldıktan sonra ayrılan yerin boyutunda değişiklik yapmak içinse realloc kullanılır. **Free değişkenin kendisini silmez, gösterdiği yeri belleğe iade eder. Java'da new ile oluşturulur fakat belleğe iade işlemini Java kendisi yapar.

```
int *p=malloc(sizeof(int));
*p=100;
printf("%d\n",*p);
free(p);
->100
```

Heapte p int olduğu için int kadarlık yer açtırdık.

***Burada p stackte bir değişkendir, = dediğimiz yerde mallocla anonim heap bellekte int boyutunda bir yer açılır ve p değişkeni oranın adresini gösterir. Dolayısıyla `printf("%x",p);` dediğimizde tutuyor olduğu heap bellekteki adresi, `printf("%x",&p);` dediğimizde ise stack bölgedeki kendi adresini yazdırır.

`int *p=malloc(sizeof(int));` bu 1 elemanlı dizi oluşturmuş olur. Örneğin 9 elemanlı bir dizi heap bellek bölgesinde oluşturmak istersek,

`int *p=malloc(9*sizeof(int));`

C'de int* dediğimizde int tipinde bir gösterici oluşturmuş oluruz. Aynı şekilde diğer tiplerde de geçerli. Fakat void* şeklinde türden bağımsız gösterici oluşturulabilir. Kutulama mantığıdır. Birden fazla türden göstericiyi bu tanımlanan void'e atayabiliriz fakat kullanacağımız esnada bunun türünü belirtmek gerekir. Örn;

```
int x=100;
void *obj;
obj=&x;
printf("%d",*(int*)obj);
->100
```

Görüleceği üzere içerisinin int olduğu bilindiği için %d ve (int*)ibaresi kullanılmış.

Aynı olay Java'da da vardır. Kullanımı object'tir ve aslında arkaplanda gizlice tüm sınıflar bu sınıftan kalıtım alır. Bu da her nesnenin aynı zamanda bir object olduğu manasına gelir. Java'da bu çok daha basittir, dönüşüm işlemine gerek yoktur.

```
Object z=60;
System.out.println(z);
z="Sakarya";
System.out.println(z);
->60
->Sakarya
```

Görüldüğü üzere Object olarak z tanımlanıyor ve önce int, daha sonra String tipinde olmasına rağmen problem yaşanmıyor.

Java ve C'de 2 int'i böleceksek ve bölündüğü değişkeni double yaparsak çıkan sonuç doğruyu vermez. Çünkü işleme giren değişkenler arasında o türden bir ifade gereklidir. Java için örn;

```
int a=7,b=2;
double c=a/b;
System.out.println(c);
->3.0
```

Fakat int'lerden birini double olarak değiştirirsek doğru sonucu alabiliriz. örn a'yı double yaparız.

Java print edilen şeyi nasıl yazabileceğini bilir, örneğin bool veri tipine sahiptir.

```
int a=5,b=4;
System.out.println(a>b);
->true
```

C'de bool olmamasına rağmen aynı işlemi %d yani int yazdırırsak 0, 1 şeklinde ifade eder.

```
int a=4,b=3;
printf("%d",a>b);
->1
```

Temel Programlama Kavramları

Değişken

özellikleri;

-isim

-adres

-değer

-tip(tür)

-yaşam süresi

-kapsam

-sabit

İsim Özelliği:

-uzunluk(eskiden 6 ila 30, 31 karakterle sınırlıymış. günümüzde hemen her dilde sınırsız)

-hangi karakterler kullanılabilir(;, ', ", gibi karakterlerin kullanılamaması durumu)

-büyük küçük harf duyarlılığı var mı(varsa okunabilirliği düşürür)

-hangisi ayrılmış kelime, hangisi anahtar kelime(fortran gibi dillerde değişken türünü değişken adı şeklinde kullanabiliyorken örn: real=5(anahtar kelime-real), c vs gibi ileri dillerde double=5 gibi bir tanımlama yapılamaz(ayrılmış kelime-double)

Tip Özelliği:

Çoğu dilde double>float>int>char'dır boyut olarak.

C'de struct olan yapıda birden fazla değişken tutabilmeyi sağlar. Struct'ın boyutunu içerisindeki

değişkenler ve sayıları belirler. Örneğin struct içinde int a, double b var ise int+double boyutu struct'ın boyutudur.

Sabit Özelliği:

İsim sabiti ve değer sabiti olarak 2'ye ayrılır.

Örneğin y=x+4'de 4 değer sabitidir ve bu şekilde kullanılması önerilmez.

const int a=4; denildiğinde bu bir isim sabiti olur ve y=x+a şeklinde kullanılması önerilir.

İşlemciler

Özelliklerine göre 2'ye ayrılırlar.

Genel Özellikleri:

- -işlenen sayısı
- -işlemcinin yeri
- -işlem önceliği
- -birleşme özelliği

Niteliklerine Göre:

- -sayısal işlemciler
- -ilişkisel işlemciler
- -mantıksal işlemciler

İşlenen Sayısı:

İşlenen demek operand demektir. Yani örneğin toplama için en az 2 operand(+,=) gerekir. Örneğin -10'da 10 sabit, - ise operanddır. Bunlar 2'li operandlara örnektir. 3'lüye örnek olarak ?:, = kullanımı olabilir.

İşlemcinin Yeri:

++x → önce arttır sonra x'i kullan demektir

x++ → önce x'i kullan sonra arttır demektir

İşlem Önceliği:

Matematikteki işlem önceliği ile aynı olmak zorundadır.

Birleşme Özelliği:

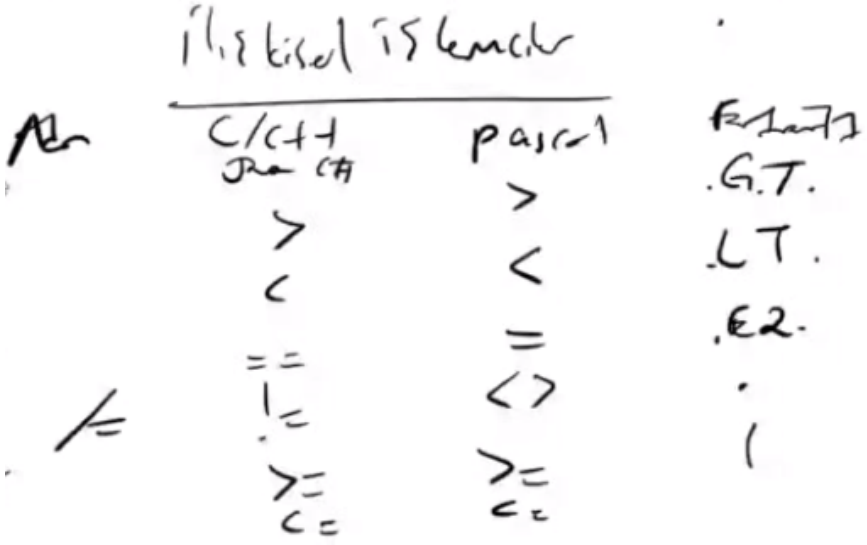
Örneğin 8/4/2 sağdan ve soldan başlandığında farklı sonuçlar verir. Normalde soldan sağa doğru gider yani sol birleşmelidir denir.

Sayısal İşlemciler:

Normal x+y gibi sayısal işlemlerdir. Tek önemli olan nokta işleme giren değişkenlerden hangisinin boyutu daha büyükse sonuç o tipte değişken üretir. Örneğin x double, y int ise x+y'nin sonucu double boyut olarak daha büyük olduğundan double olur. int a=x+y bile olsa x double y int ise x+y yani sağ taraftan double bir değer gelir fakat bunu int a'ya attığımızda C'de değer kaybı yaşarız, Java'da olsaydı hata verirdi çünkü dönüştürmemizi beklerdi. Yani mühim olan sağ taraftan double geldiğini bilmektir.

İlişkisel İşlemciler:

C/C++, Java, C# gibi aynı ailede olan dillerde >,<,==,!=,>=,<= gibi ifadelerdir.



Örneğin a+5>6 bir ilişkisel işlemcidir ve bu True veya False döndürür.

Mantıksal İşlemciler:

Daha çok ilişkisel işlemciler içerisinde kullanılırlar. &&-and, ||-or, !-not vb gibi kullanımlardır. Bool karşılaştırma yapar, yani x=a && y=b sağ ve solun True gelmesi gerekir.

Bazı diller(çoğu) mantıksal işlemcilerin kullanımında kısa devre değerlendirme kullanılır. Örneğin 6 adet koşul birbiri ile && ile bağlandıysa ve 1. koşulda False geldiyse hepsi && kullandığı ve True gelmesi gerektiği için teki False olunca diğerlerine bakmaz. || içinse ilki True gelse diğerlerine bakmaz.

True veya False döndürür.

Öncelik Sıralaması;

Sayısal İ>İlişkisel İ>Mantıksal İ.

Operatör, İşlemci Aşırı Yükleme (Operator Overloading):

Örneğin aslında 2 string toplanamaz fakat + operatörü 2 stringi birleştirmesi gerektiğini bilir. Aynı şekilde A sınıfından türetilmiş a1 ve a2 nesnesi de a1+a2 şeklinde nasıl bu toplama işlemini gerçekleştireceğini bilmediği için toplanamazlar. Bu durumda biz + operatörüne bu 2 nesneyi nasıl toplayacağını overloading yaparsak bu işlemi yapabiliyor hale gelir. Örneğin a1+a2 yani direkt nesne isimlerini yazdığımızda a1 ve a2'nin yaş değişkenlerini birbiri ile topla diyebiliriz.

Bu özellik C'de yoktur. Java dilinde de yoktur. Bu özellik low level bir özelliktir. C++ ve C#'da vardır.

Atama Operatörü:

Atama işlemcisi, operatörü sağ taraftaki ifadeyi soldakine atamaya yarar. Örn a=8; Sol tarafa herhangi bir değer sabiti yani a yerine 100 vs yazılamaz. C vs. dillerde = iken, bazı dillerde :='dir.

x,y=0; dediğimizde hem x hem y'ye 0 atanır. Buna çok hedefli atama denir.

x=y=0; şeklinde de kullanılır. Sağdan sola doğru ilerler. Yani önce y'ye 0 atar ve sonrasında ise x'e y'nin değerini yani 0'ı atar.

Bileşik Atama:

+=, -=, /= gibi atama operatörleridir. Ara koda dönüşüm esnasında bu operatörü düzenleyerek anlayabileceği şekle çevirir.

unsigned → C, C++, Ada+ C# gibi dillerde işaretsiz bir sayıyı ifade eden tiptir. Yani negatif bir sayı tanımlanamaz. Java'da bu yoktur.

Kayan Noktalı Sayı: (Floating Point)

Ondalıklı sayı demektir. Float'ın çıkışı buradandır.

C#'da float ve double'a ek olarak decimal veri tipi de vardır.

Yaklaşık olarak float virgülden sonra 7, double 14-15, decimal 32 sayıya sahiptir.

Set (Küme):

Listeden bakış açısı olarak farklıdır. Örneğin listede regexlerle vs. ifade edebileceğimiz rakamlar set ile çok daha basittir.

Rakam = set of [0..9] şeklinde kolayca küme yapısı tanımlanabilir.

Dizilerin adresi dizinin 1. elemanının adresi olarak tutulur. Dizinin örneğin 3. elemanının adresini bulmak için dizinin 1. elemanının adresine dizinin tipi boyutu*bulunduğu indeks sayısal olarak eklenirse bulunur. Tek boyutlu dizi için Örn:

`int a[]={10,20,30};` şeklinde tanımlanmış bir dizide a dizisinin adresi a[0]'ın adresi yani 10'un adresidir. Eğer 30'un adresini veya herhangi i. indeksteki elemanının adresini bulmak istiyorsak;

Adres(a[i]) = Adres(a[0]) + i*c 'dir.

c int'in boyutu 4 byte olduğu için 4'tür. Yani 30'u bulacaksak ve dizi int olarak tanımlandıysa,

Adres(a[2])=Adres(a[0])+2*4'tür. Dizi indekslerine anlık erişim de bu şekildedir.

Hafızada

10

20

30

şeklinde tutulur.

2 boyutlu dizi için Örn:

`int a[][]={10,20,30},{25,15,2};` şeklinde bir dizi tanımlansa

burada 2. parantez yani 25 15 2 2. satırı temsil eder. Yani şu an aslında a dizisi [2][3] şeklinde bir dizidir. 2 satır ve 3 sütundur. Örneğin 20 sayısı [0][1]'de bulunur. Yine tek boyutludaki adres tutma mantığıyla aynı olup, ilk elemanı yani [0][0]'ı adres olarak tutar.

Adres(a[i][j]) = Adres(a[0][0]) + ((i*n)+j)*c 'dir.

burada "n" sütun sayısını temsil eder. Bu örnek için n 3'tür.

Hafızada

10

20

30

25

15

2

şeklinde tutulur. Formülün mantığı da buradan gelmektedir.

Mühim olan burada adresler 16'lık hexadecimal tabanda tutulurlar. Yani alacağımız tür boyutunu da o tabanda ekleme yapmamız gerekir.

C ve Java'da if'ler süslü parantezsiz iç içe yazılabilir.

```
int x;
printf("Bir sayi giriniz: ");
scanf("%d",&x);
if(x%2==0)
  if(x<100)
    if(x>10)
      printf("Girilen sayi 10'dan buyuk, 100'den kucuk ve cift!");
```

Burada hata vermez çünkü içerideki ifler bir önceki if'in True dönmesi sonucu çalışırlar. Fakat herhangi bir else konulduğunda tab sayısı fark etmeksizin en

sondaki if'e ait olur. İlk if'in elsinin olması isteniyorsa x%2==0 işleminden sonra süslü açılıp printf'den sonra süslü ile kapatılması gerekir. Bu yüzden kod tek satırlık bile olsa süslü ile kullanılması tavsiye edilmektedir.

Switch-Case yapısı için bir değişken belli sayıda değer kontrol edilecekse, bir aralık kontrol edilecekse if kullanılabilir. Genel olarak kullanım farkı budur.

Java'da switch-case'de herhangi bir tipte değişken kullanılabilir fakat c'de sadece tam sayılar desteklenmektedir.

Javada Scanner in=new Scanner(System.in) ile konsoldan okuma yapılır. System.in konsolu ifade eder.

```
Scanner in=new Scanner(System.in);
System.out.println("Ulke: ");
String ulke=in.next();
```

Üçlü operatör kullanımına örnek olarak

`(sayi%2==0 ? "Sayi Cifttir" : "Sayi Tektir");`

burada eğer sayinin 2'ye bölümünden kalan 0 ise yani koşul True ise Sayi Cifttir, değil ise (":") Sayi Tektir yazar. Burada dikkat edilmesi gereken ":" sağı ve solu aynı tipte olmalıdır. (string bu örnek için char*)

C#'da ?? operatörü bulunmaktadır. Bu operatör NULL değil ise kullan, NULL ise yeni nesne üret şeklinde kullanılır. Örn;

this.veri = veri ?? new Sayi();

şeklinde kullanılır. Yani eğer veri NULL değilse this.veri'ye ata, NULL ise Sayi sınıfından bir nesne türet demektedir.

Bağlama Kavramı

Bir özellik ile program elemanı arasında ilişki kurulmasına bağlama kavramı denir. Örneğin isim ile değişken türünün bağ kurması. Programın tasarlanmasında, derlenmesi aşamasındaki bağlama kavramlarına statik, programın çalışma anında bağlama kavramlarına dinamik bağlama kavramı

denir. Örneğin = atama operatörü dilin tasarlanması aşamasında yapılmıştır ve statik bir bağlama kavramıdır.

Statik Tip Bağlama

1-Örtülü Tip Bağlama

Burada belirteç kullanılmaz yani int, double, char gibi. Örneğin basic dilinde son karakteri $ olan bir değişken char tipi olarak ifade edilir. Yani biz bu ifadeyi aslında daha program çalışmadan yazmışız, derlenme anında anlıyor. Yani biz buna char dememişiz ama $ ile bittiği için char olduğunu anlıyor. Buna örtülü statik tip bağlama deniyor. Fortran'da ise isim I J K L M ile başlıyorsa int demektir. Günümüz dillerinde buna çok rastlamayız.

2-Dışsal Tip Bağlama

Türün ne olduğunun belirtildiği(int, float, char vs.) yani günümüz programlama dillerinde kullandığımız tip bağlamadır. Bu bağlama tipi güvenlik nedeniyle tercih edilmektedir.

Dinamik Tip Bağlama

Daha çok yorumlayıcılı dillerde rastlanır. Örneğin a=12.4 dediğimizde bunu double ile bağlar. Güvenliği zedeleyen bir özelliktir. Kısmen derleyicili dillerde de rastlarız.

Örn diyelim ki Java'da Canli isminde bir class tanımladık (içerisinde kos isimli bir metot oldugunu varsayalım) ve sonrasında Insan ve Hayvan diye sınıflar tanımlayıp Canli sınıfından kalıtım aldık.

Sonrasında main içerisinde `Canli c=new Insan();` dedik, İnsan gibi görünen bir canlı oluşturduk. Bu dinamik tip bağlamaya bir örnektir.

Görünümlü nesne üretmeye örnek olarak;

Örneğin Canli sinifinda kos metodu var ve Canli sınıfından kalıtım alan Insan içerisine de kendine ait olan konus metodu tanımladık.

`Canli c=new Insan();` dersek c.konus() diyemeyiz çünkü Canli sınıfı böyle bir metoda sahip değil. `Insan c=new Insan();` şeklinde tanımlamalıyız ki tam olarak bir

İnsan türetelim ve konus metodunu kullanabilsin.

Eğer ki Insan içerisinde Canli sınıfından bulunan bazı metotları override ederek yeni tanımlamalar yapmış olsaydık (mesela kos metodu) `Canli c=new Insan();` şeklinde de tanımlamış olsak yine Insan'da bulunan override edilmiş olan metotlar çağırılırdı.

Bellek Bağlama

Bir değişkenin erişilebilir bir bellek hücresi ile ilişkilendirilmesi işlemine denir. Bellekte her hücrenin Adres, İsim ve Değer kısımları vardır. Bir bellek hücresinde bir değişkenin tutulma süresine ise **yaşam süresi** denir.

Ram Belleğin görünümü şu şekildedir; RTS → RunTime Stack

.exe dosyasına basıp çalıştırdığımızda derlenmiş kod olarak bellekte tutulur.

Statik Bellek Bölgesi

Bazı programlama dillerinde desteklenmez bile. Bu bölgede global ve static local değişkenler bulunur. Global değişken emir esaslı dillerde bulunur ve kod içerisinde her yerden erişilebilir. Static bellek bölgesinin temel özelliği program başladığı an değişkenin yaşam süresinin başlayıp, sonlanmasıyla bitmesidir.

Ada bu bellek bölgesini desteklemez.

Fortran77'de lokal değişkenler bu bölgede tutuluyordu. Şu an RTS'de.

RTS

Bütün olay burada dönüyor. Lokal değişkenler, fonksiyon parametrelerinin bulunduğu yerdir. Bir fonksiyon çağırıldığında bunun bir **aktivasyon kaydı** oluşur ve fonksiyonla ilgili değişken, parametreler ve geri dönüş adresi burada oluşur. Fonksiyon (bu for-while da olabilir) kapandığı yani return yaptığı an o fonksiyonla ilgili her şey silinir. Dolayısıyla RTS'deki yaşam süresi o lokal bölgenin aktif olduğu zaman kadardır. Örn:

```
for (...){
    int a;
    ...}
```

şeklinde bir for döngüsü tanımlı ve a değişkeni içerde tanımlanmış. Dolayısıyla bu for döngüsü başlayıp bittiğinde o a değişkeni silinmiş olur.

Heap Bellek Bölgesi

Yaşam süresini bizim belirleyebildiğimiz bölgedir. Bellek hücresinde bulunan Adres, İsim ve Değer'den ismi heap bellek bölgesinde kaybederiz. Bu yüzdendir ki heap bellek bölgesindeki elemanlarla iş yaparken isimle değil adreslerle uğraşırız. Gelişmiş dillerde bunu bir nebze kolaylaştırmaya yönelik referans olayı vardır. Örneğin Java'da `Kisi k=new Kisi();` dediğimizde k aslında RTS'de bir değişkendir fakat adres olarak gösterdiği yer Heap bellek bölgesindedir. Yani k new Kisi() ile heap bellek bölgesinde oluşturduğumuz bellek hücresinin bir referansıdır. k nesnenin kendisini temsil etmiyor, referansı. Bu işlem sonrası `Kisi a=k;` dersek a isimli yeni bir bölge oluşturmaz. Burada yaptığı iş yine a RTS bölgesinde bir değişkendir fakat gösterdiği yer k'nın gösterdiği yerdir.

new kelimesi Heap bellek bölgesinde yer oluşturmak anlamında gelir. Yani new olan yerde dönen şey her zaman adrestir.

C dilinde new, malloc'tur.

Heap bellekte açılan bu yeri boşalltmak için C'de free, C++'da delete, Pascal'da dispose komutları bulunur. Java, C#, Ada gibi dillerde bu işi çöp toplayıcı kendisi

belli aralıklarla belleği tarayarak yapar. Fakat bu da programı aslında yavaşlatan bir şeydir.

Örtülü Heap Değişkenler

Mesela JavaScript'de bir dizi tanımlanıyorsa bu heap bellek bölgesinde tanımlanır. Yani dizi tanımlama esnasında heap bellek bölgesinde oluşturma komutu new, malloc gibi ifadeler kullanmadan direkt diziAdi=[1,2,3] şeklinde tanımlanan dizi örtülü olarak heap bellek bölgesinde oluşur. Yine bu işi de genelde yorumlayıcılı diller yapar.

İsim Kapsamları

Bellekte bulunması onun erişilebilir olduğu anlamına gelmez. Kapsam dahilinde değilse erişilemez. Aynı isimde bir çok değişken aynı anda bellekte bulunabilirler. Bunun kapsamını scope'lar belirler. Fakat iş nesne yönelimli bir dile geldiğinde scope'a ek olarak erişim belirleyiciler olan private, publicler de devreye girer.

Statik(Durağan) Kapsam Bağlama

Program kodunu yazdığımız metinsel düzene göre belirlenen bağlamlardır. C dili statik kapsam bağlama kullanır fakat sınavda soruda c kodu verilip dinamik kapsam bağlamaya göre çöz denirse o şekilde çözülür.

Örn(Pascal):

```
Program L;
  var n:char;
  Procedure W;
  begin
    write(n)
  end
  Procedure D;
    var n:char;
  begin
      n:="D";
      W;
  end
begin
n:="L";
W;
```

```
D;
end
```

en alttaki begin end Program L'nin gövdesidir. Ekran çıktısı ne olur?

`n:="L";` hangi n'ye "L"yi atıyor? Burada n değişkeni L programına bağlı bir değişken. Dolayısıyla program başında ana olarak `var n:char;` (Buna 1.n diyelim) olarak tanımlanmış n'ye "L" ataması yapıldı.

`W;` prosedürü çağırıldığında blok içinde n'yi ekrana yaz diyor. Burada ne yazacak? W prosedürü içerisinde herhangi bir n değişkeni tanımlaması yapılmadığından dolayı otomatik olarak bir üst aralıkta n'yi aramaya başlayarak o n'yi ifade ettiğini anlıyor. Yani bu da Program L'nin içerisine denk geliyor ve "L" yazdırıyor.

`D;` prosedürü çağırıldığında blok içinde `var n:char;` (Buna 2.n diyelim) olarak bir n isimli değişken tanımlanmış. n'ye "D" ata denmiş, hangi n'ye atayacak? Kendi bloğu içerisinde tanımlanmış olan yeni, yani 2.n'ye "D" ataması yapılıyor. Sonrasında yine D prosedürü içerisinde W prosedürü çağırılıyor ve W prosedürü n'yi yazdıracak fakat içerisinde n isimli bir değişken yok. Dolayısıyla bir üste geçiyor ve Program L'de olan 1.n değişkenini görüp ekrana "L" yazdırıyor.

Burada dikkat edilmesi gereken, D prosedürü W prosedürünü çağırdı fakat W prosedürü kendisini kimin çağırdığını sorgulamadı. Bu yüzden de aslında kendisini çağıran D prosedürünün n değişkenini yani 2.n değişkenini yazmak yerine bir üst bloğunda bulunan n değişkenini yani 1.n'yi yazdı. Statik ile dinamik kapsam bağlamanın farkı buradadır.

Bu programın **statik kapsam bağlamaya göre çıktısı LL** olur.

Örn(C):

```
int x;
int main(){
  x=2;
  f();
  d();
}
void f(){
  int x=3;
  h();
}
void d(){
```

```
  int x=4;
  h();
}
void h(){
  printf("%d",x);
}
```

Öncelikle en başta tanımlanmış olan x'e(1.x) 2 ataması yapılıyor. Ardından f fonksiyonu çağırılıyor ve yeni bir x(2.x) değişkeni tanımlanıp 3 ataması yapılıyor. Burada h fonksiyonu çağırılıyor ve h fonksiyonu x'i yazdıracak. Hangi x'i? Kendi bloğuna bakıyor x yok dolayısıyla bir üst bloğa yani kaynak kodun tamamında olan 1.x'i 2'yi yazdırıyor. f fonksiyonu kapanmış oluyor ve f fonksiyonu RTS'de çalıştığı için içerde tanımlanan 2.x yok oluyor. Yani şu an tek x var. d fonksiyonu çağırılıyor ve yine aynı işlemlerle 2 yazdırılıyor. Ardından d'de oluşturulan x değişkeni de yok oluyor.

Bu programın **statik kapsam bağlamaya göre çıktısı 22** olur.

Dinamik Kapsam Bağlama

Çalışma zamanında belirlenen bağlamlardır. Yani çağırılan prosedür, fonksiyon kendisini çağıran yerdeki değişkeni baz alarak işlem yapar. Çünkü çalışma zamanında nereden çalıştırıldığını bilir. Dinamiğin olayı budur.

Örn(Pascal):

```
Program L;
  var n:char;
  Procedure W;
  begin
    write(n)
  end
  Procedure D;
    var n:char;
  begin
      n:="D";
      W;
  end
begin
n:="L";
W;
D;
end
```

en alttaki begin end Program L'nin gövdesidir. Ekran çıktısı ne olur?

`n:="L";` hangi n'ye "L"yi atıyor? Burada n değişkeni L programına bağlı bir değişken. Dolayısıyla program başında ana olarak `var n:char;` (Buna 1.n diyelim) olarak tanımlanmış n'ye "L" ataması yapıldı.

`W;` prosedürü çağırıldığında blok içinde n'yi ekrana yaz diyor. Burada ne yazacak? W prosedürü içerisinde herhangi bir n değişkeni tanımlaması yapılmadığından dolayı kendisini çağırmış olan Program L'ye ait olan n'yi ifade ettiğini anlıyor. Yani "L" yazdırıyor.

`D;` prosedürü çağırıldığında blok içinde `var n:char;` (Buna 2.n diyelim) olarak bir n isimli değişken tanımlanmış. n'ye "D" ata denmiş, hangi n'ye atayacak? Kendi bloğu içerisinde tanımlanmış olan yeni, yani 2.n'ye "D" ataması yapılıyor. Sonrasında yine D prosedürü içerisinde W prosedürü çağırılıyor ve W prosedürü n'yi yazdıracak fakat içerisinde n isimli bir değişken yok. Dolayısıyla W prosedürü kendisini çağıranın kim olduğuna bakıyor, D. D prosedürüne baktığında n'nin yani 2.n'nin tanımlı olduğunu görüyor ve 2.n'yi ekrana "D" yazdırıyor.

Bu programın **dinamik kapsam bağlamaya göre çıktısı LD** olur.

Örn(C):

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

Öncelikle en başta tanımlanmış olan x'e(1.x) 2 ataması yapılıyor. Ardından f fonksiyonu çağırılıyor ve yeni bir x(2.x) değişkeni tanımlanıp 3 ataması yapılıyor.

Burada h fonksiyonu çağırılıyor ve h fonksiyonu x'i yazdıracak. Hangi x'i? Kendi bloğuna bakıyor x yok dolayısıyla kendisinin çağırıldığa bloğa giderek orada x arıyor ve 2.x olan 3'ü yazdırıyor. f fonksiyonu kapanmış oluyor ve f fonksiyonu RTS'de çalıştığı için içerde tanımlanan 2.x yok oluyor. Yani şu an tek x var. d fonksiyonu çağırılıyor ve yine aynı işlemler ve mantıkla 4 yazdırılıyor. Ardından d'de oluşturulan x değişkeni de yok oluyor.

Bu programın **dinamik kapsam bağlamaya göre çıktısı 34** olur.

C'de foreach yoktur. Java'da da yoktur fakat koleksiyon içerisinde dolaştırmaya yarayan farklı bir yapı bulunmaktadır.

`for(int i : x)` x dizisinin eleman sayısı kadar döner ve her defasında içerisindeki değeri i'ye atar.

İç içe döngülerde çarpan etkisi vardır. Örneğin 100 kez dönen bir döngünün içerisine 10 kez dönen bir başka döngü koyarsak bu 1000 defa çalışacaktır.

Durum etiketleri;

continue → bulunduğu blok içerisindeki iterasyonu sonlandırarak bir sonraki iterasyona geçer.

break → bulunduğu döngü bloğunu, scope'unu sonlandırır.

Örn:

```
for(int i=1;i<=9;i++){
  System.out.print(i+" |");
  for(int j=1;j<=9;j++){
    if(i*j<10) System.out.print("   "+i*j);
    else break;
}}
```

kodunda i*j 10 ve büyük olma durumunda break ile koşuldan çıkması sağlanıyor. Bazen en içteki döngüde bir koşuldan sonra en dıştaki for'dan çıkmak isteyebiliriz. Bu örnekteki break ait olduğu scope içerisini yani j'li for'u

sonlandırır. Eğer ki biz bu durum olduğunda istediğimiz herhangi for'u (for'un kaç kez iç içe olduğu fark etmez, 5 tane iç içe for'dan 5. fordan break'i en üstten 2.sine kadar da break atabiliriz.) break ile sonlandırabilmemizi Java outer ile sağlayabiliriz.

```
outer:
for(int i=1;i<=9;i++){
  System.out.print(i+" |");
  for(int j=1;j<=9;j++){
    if(i*j<10) System.out.print("   "+i*j);
    else break outer;
}}
```

break atmak istediğimiz döngü başına `outer:` ekler ve break attığımız yerde yalnızca break yazmak yerine `break outer;` yazarsak outer:'ın olduğu yerden itibaren break atarak oradan çıkmış oluruz.

C'de buna benzer bir yapı bulunmamaktadır fakat goto ile aynı iş görülebilir.

```
for(int i=1;i<=9;i++){
  printf("%d | ",i);
  for(int j=1;j<=9;j++){
    if(i*j<10) printf("   %d",ij);
    else goto outer;
  }}
outer:;
```

şeklinde for'un ilerisine atlanarak aynı işlem yapılabilir fakat kesinlikle tavsiye edilmez!

Yapısal Programlama

Sıralı yapılar

Kodun yukarıdan aşağı doğru okunması.

Şartlı yapılar

if blokları vs.

Yinelemeli yapılar

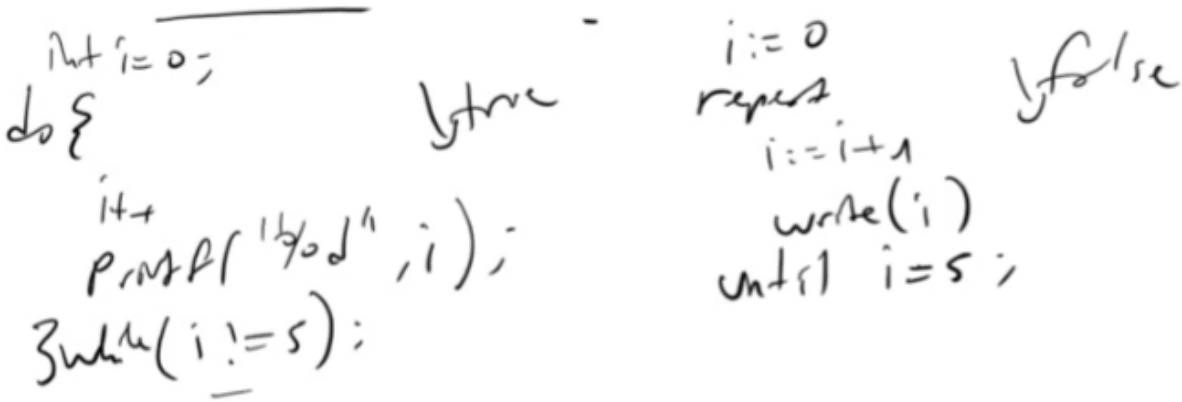
Sayaç kontrollü döngü yapıları → for, foreach durumları

Mantıksal kontrollü döngü yapıları

Pretest(önce kontrol) → while, for (döngü hiç çalışmayabilir durumu)

Posttest(sonra kontrol) → do-while, repeat until veya until(lisp ve pascal) (döngü en az 1 defa çalışır)

Do-While ve Repeat-Until kullanım farkı



do-while true olduğu sürece, repeat-until false olduğu sürece çalışır.

Modüler Programlama

Mantığı sınıfa yalnızca tek sorumluluk yüklemektir.

Fonksiyon bir şey döndürüyorsa fonksiyon, döndürmüyorsa metot demek doğru olandır.

`int topla(int a, int b)` gibi bir fonksiyonda int a, int b **formal parametre**lerdir. Yani gerçek değillerdir çünkü onlara asıl değerleri dışarıdan verilecektir. Main içerisinde `topla(a,b)` dediğimizde gönderdiğimiz a,b ise **gerçek parametre**lerdir.

Aktivasyon Kaydı

Bir fonksiyon çağırıldığında RTS bellek alanında oluşturulur. Burada sırasıyla

Geri dönüş adresi → A() içinde B() çağırılırsa B()'nin geri döneceği adresi bilmesi gerekir o yüzden

Statik kapsam bağlama için link → değişken varsa kontrol etmesi gerekir o yüzden

Formal parametreler

Lokal değişkenler

tutulur ve fonksiyon bittiğinde aşağıdan yukarı geri dönüş adresini görene kadar gelir ve o aktivasyon kaydını siler.

Örn:

```
int toplam(int a, int b){
  int sonuc=a+b;
  return sonuc;
}
int main(){
  int x=10;
  int y=20;
  toplam(x,y);
}
```

gibi bir kodda

Geri dönüş adresi

Statik kapsam bağlama için link

a 10 | formal parametreler (x'den değeri aldı(x main fonksiyonunun aktivasyon kaydından geldi))

b 20 | formal parametreler (y'den değeri aldı(y main fonksiyonunun aktivasyon kaydından geldi))

sonuc 30 | lokal parametreler

gibi tutulur.

Parametre Geçirme Yöntemleri

Çağırandan çağırılana neyin gittiğidir.

Değer ile Çağırma

Çağırandan çağırılana değer gider.

Değer ile çağırmada formal parametre tanımlanır.

Aktivasyon kaydı örneğindeki gibidir.

Adres ile Çağırma

Değer ile çağırmadan tek farkı değer yerine adres gönderilmesidir, yani parametre pointer olacak.

Yani aktivasyon kaydı örneğinde a ve b yerine *a *b olması ve main içerisinde x ve y'nin int *x=&q1; ve int *y=&q2; olarak tanımlanarak toplam(x+y) gönderilmesi gibi.

**Ek Bilgi: Burada a ve b yine bir kopya misalidir fakat *& şeklinde çağırım yapsaydık bu durumda a ve b fonksiyonu çağırırken gönderdiğimiz parametrelerin kendileri olacaktı. Yani böyle bir çağırım yaparsak bu durumda fonksiyon içerinde eğer a'nın veya b'nin gösterdiği yeri değiştirdiğimiz durumda gönderdiğimiz parametre olan pointer'ın da gösterdiği yer değişmiş olur.

https://www.youtube.com/watch?v=7HmCb343xR8

Adres ile çağırmanın Java örneği;

```
void guncelle(Ogrenci o1){
  o1.isim="Ahmet";
}
int main(){
  Ogrenci o2=new Ogrenci("Ali");
  guncelle(o2)
}
```

Yani mainde o2 nesnesi oluşturulduğunda heapte bir adresi gösterecektir ve sonrasında güncelle metoduna bunu gönderdiğimizde o1 isimli parametre de o2'nin gösterdiği adresi gösterecektir. Dolayısıyla o2 öğrencisinin ismi Ali'den Ahmet olarak değişecektir.

Referans ile Çağırma

Çağırılanın muhakkak etkilendiği bir türdür. Formal parametre yoktur yani alınan parametre fonksiyonun içerisinde bir kopya olmayacaktır, içeride yapılan her işlem parametre olarak gönderilen değişkeni etkileyecektir. C ve Java desteklememektedir. Java'da referans ile çağırma denilen şey aslında adres ile çağırmadır!

```
void guncelle(int &x){
  x=100;
}
int main(){
  int a=5;
  guncelle(a);
  cout<<a;
}
->100
```

`(int &x)` buradaki işlem bunu referans ile çağırma yapan yerdir. Yani x için bir yer ayrılmamıştır, yalnızca gelecek olan değerin bir referansıdır. Yani bu örnek için x aslında a'nın referansıdır ve yalnızca bir takma isimden ibarettir. Aslında guncelle içerisinde her x denildiğinde aslında a demek olmuş olur.

Burada anlaşılması gereken en önemli yer;

Adres ve değer ile çağırmada fonksiyon içerisinde yapılan işlemler formal parametre üzerinde yapılır ve gelen değer kesinlikle etkinlenmez! Örn adres ile çağırma (*) örneğinde guncelle içerisinde o1=NULL; dersek, gelen o2 nesnesinin gösterdiği adres değil, o1'in gösterdiği yer NULL olur. Yani aslında parametre olarak o2 geldiğinde formal parametre olan o1, o2'nin gösterdiği yeri gösteriyordu ve biz bunu o1'in gösterdiği yer NULL olsun olarak güncelledik. Yani guncelle metodundan çıktıktan sonra o2'nin ismi yazdırılmak istendiğinde yine Ali'yi yazar!

Referans ile çağırmada (&) ise durum böyle değildir, o1=NULL dersek o2'nin gösterdiği yer de NULL olur!

C#'da void guncelle(ref int x) ve çağırılırken de guncelle(ref a) şeklinde aynı işlem yapılabilir. Bir diğer versiyonu in int x şeklindedir, in'de yalnızca okuma yapılabilir fakat refte fonksiyon içerisinde değişiklik de mümkündür.

Sonuç ile Çağırma

Gönderilen parametrenin fonksiyon içerisinde anlam kazandığı türdür. C ve Java'da bulunmamaktadır.

C#'da bunun için out kullanılır. ortHesapla fonksiyonu tek 1 parametre alarak çalışır. Yani aşağıdaki örnek için aslında fonksiyonun asıl işlevi aldığı dizinin ortalamasını döndürmektir. Fakat biz bu dizinin maksimumunu da bilmek istiyorsak böyle bir şey yaparız. bu kullanımda maks içeride en az 1 defa kullanılmak zorundadır yoksa derlenme hatası alır.

```
double ortHesapla(int []dizi, out int maks)
```

C++'da bunu benzetme yoluyla `double hesapla(int []dizi, int &maks)` ile yapabiliriz.

İsim ile Çağırma

Parametre olarak fonksiyon adının gelme durumudur.

JavaScript ve Algol60'da vardır. Daha çok yorumlayıcılı dillerde görülen bir türdür.

Örneğin harfleriHesapla, AFHesaplama ile AHHesaplama fonksiyonları mevcutsa

harfleriHesapla(hesaplamaYontemi) şeklinde çağırılabilir. hesaplamaYontemi yerine yazılacak şey AFHEsaplama veya AHHesaplama'dır. sonrasında harfleriHesapla fonksiyonu içerisinde hesaplamaYontemi() çağırılarak aslında oraya yazılan hesaplama yontemi (AF veya AH) çalıştırılmış olacaktır.

Özyineleme

Sürekli RTS bellek alanında aktivasyon kaydı alanı oluşturur.

Dolaylı özyineleme de mevcuttur. A→B→C→A→B.... durumudur, tehlikelidir.

Amacı karmaşık çözümleri birkaç satırda halletmektir fakat yavaşlık getirir çünkü belleği çok sık kullanır.

C ve C++'da fonksiyonlar değişken gibi davranır. Yani yukarıdan aşağı doğru okur. Fonksiyonu mainin üzerinde değil de altında tanımlayıp main içerisinde çağırmak istersek o fonksiyonu bulamaz.

Bu durum için bu dillerde eğer fonksiyonlar aşağıda tanımlanıyorsa main yukarısında prototipi yani gövdesiz tanımlamaları yapılır.

Örneğin main altında;

```
double Topla(double x, double y){
  return x+y;
}
```

şeklinde tanımlanmış Topla fonksiyonu main üstünde de prototip olarak `double Topla(double, double);` şeklinde tanımlamak gerekir.

Java'da parametre yerinde `void Fonk(int... x)` şeklinde x dizi olarak kabul edilerek değişken sayıda parametre tanımlaması yapılabilir.

https://www.youtube.com/watch?v=xPojVqfdC24

Java'da kütüphane dosyaları .jar uzantılıdır.

Java'da kütüphane oluşturma ve entegre → 6. hafta sonu

Java'da aynı paket içerisinde olan tüm sınıflar birbirleriyle haberleşebilirler. Yani protected mantığıdır.

Java'da super ile base sınıfı ima edebiliriz. C#'da bu base, C++'da çoklu kalıtım olduğu için sınıf adıyladır.

Çoklu kalıtım **Diamond Problemi**ne sebebiyet verebilir. Diamond problemi 2 veya daha fazla sınıftan kalıtım alan sınıftan türetilen nesnenin kalıtım aldığı sınıflarda bulunan bir metodu çağırdığında hangi base sınıftakini çağıracağını bilemediği durumdur.

C ve C++'da heap bellek bölgesinde bir şey oluşturmak için pointer(*) kullanmak zorunludur!!

Java'da iç içe sınıf tanımlaması yapılabilir. Tek avantajı içeride tanımlanan sınıfın üstünde bulunan sınıfın private alanlarına erişebilmesidir.

Java'da `protected void finalize() throws Throwable {..}` şeklinde yıkıcı metot oluşturulabilir. Fakat C/C++'daki gibi silinmeyi garanti etmez. Mantığı ve işleyişi biraz farklıdır.

*******C'de struct ile nesne yönelimliye benzetme yapılabilir. Bkz → Hafta 7

Interface

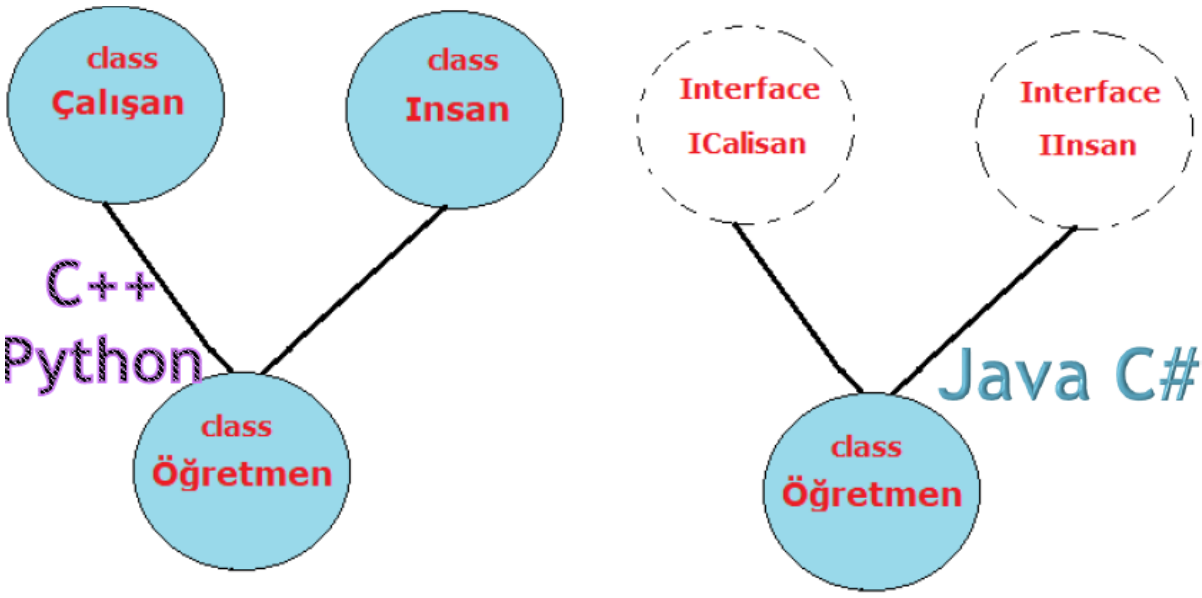
Arayüz kullanımı aslında bir sözleşme imzalamaktır.

Bir arayüzden kalıtım alan bir sınıf o arayüzü gerçekleştireceğini vadediyor demektir.

Java ve C#'ta arayüzler alan içeremez sadece public metot tanımı içerebilirler. Java ve C# dilinde direkt desteği bulunan arayüzler C++ dilinde soyut sınıflar kullanılarak gerçekleştirilebilir.

```
public interface MuzikCalar{
  void Oynat(String dosyaTuru,String dosyaAdi);
}
```

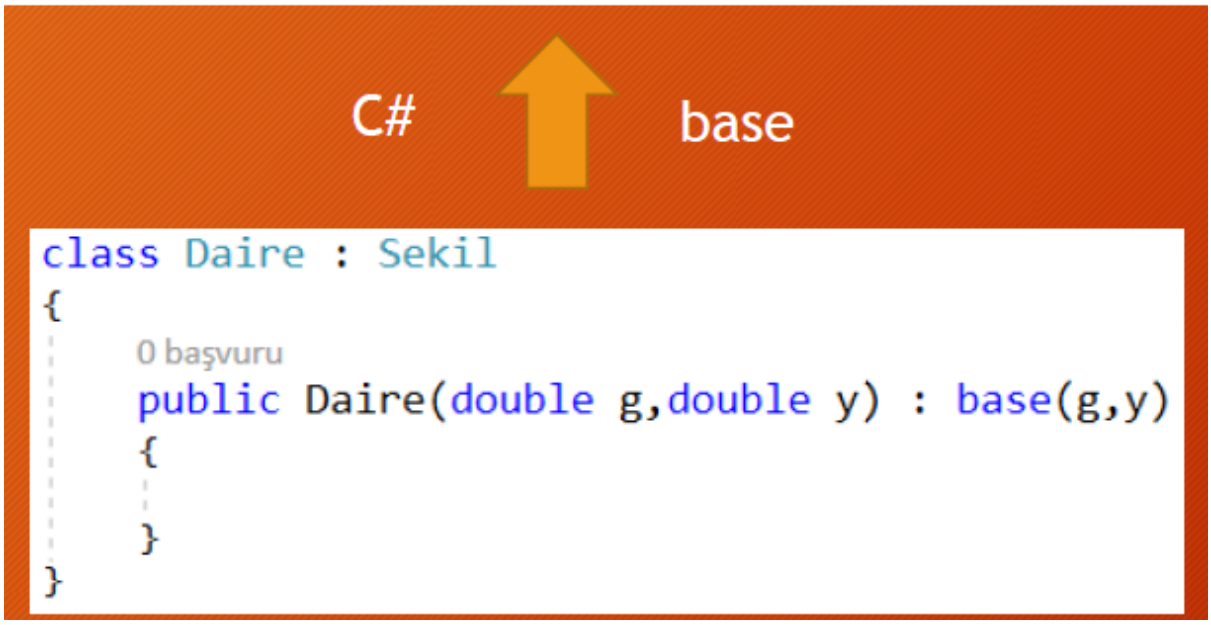
Çoklu sınıf kalıtımı izin verilmeyen dillerde çözüm olarak arayüz bulunmalıdır. Çoklu arayüz kalıtımı yapılabilir.



Sağ taraftaki örnek için çok daha fazla metot yazmak gerekecektir. Çünkü kalıtım alacak olsak kalıtım aldığımız base sınıftaki metotları yeniden yazmamıza gerek kalmayacaktı fakat Interface'den arayüz kalıtımı yaptığımızda gövdesi olmayan fonksiyonları doldurmamız gerekecek.

Kalıtım alındığında kalıtım alan sınıftan bir nesne türetildiğinde arka tarafta base sınıftan da nesne türetilmiş olur. Kalıtım alınan sınıfın kurucusu parametre bekliyor ise C#'da bunu :base() ile ifade etmek gerekir.

```
class Sekil
{
    protected double genislik;
    protected double yukseklik;
    1 başvuru
    public Sekil(double genislik,double yukseklik)
    {
        this.genislik = genislik;
        this.yukseklik = yukseklik;
    }
}
```



C++ ve Python gibi çoklu sınıf kalıtımı olan dillerde ise base sınıflar farklı parametreler alabileceği için bu durum farklı şekilde işler. C++'da bu durum için :sinifAdi(), sinifAdi() kullanılır.

```cpp
class Insan{
    protected:
        double boy;
    public:
        Insan(double boy) { this->boy = boy; }
};
class Calisan{
    protected:
        int Id;
    public:
        Calisan(int Id){ this->Id = Id; }
};
class Ogretmen : public Insan, public Calisan{
    public:
        Ogretmen(double boy,int Id) : Insan(boy),Calisan(Id){
        }
};
```

C++

Python'da ise super() ile yapılır.

```python
class Insan:
    ...
class Calisan:
    ...
class Ogretmen(Insan, Calisan):
    super()
```

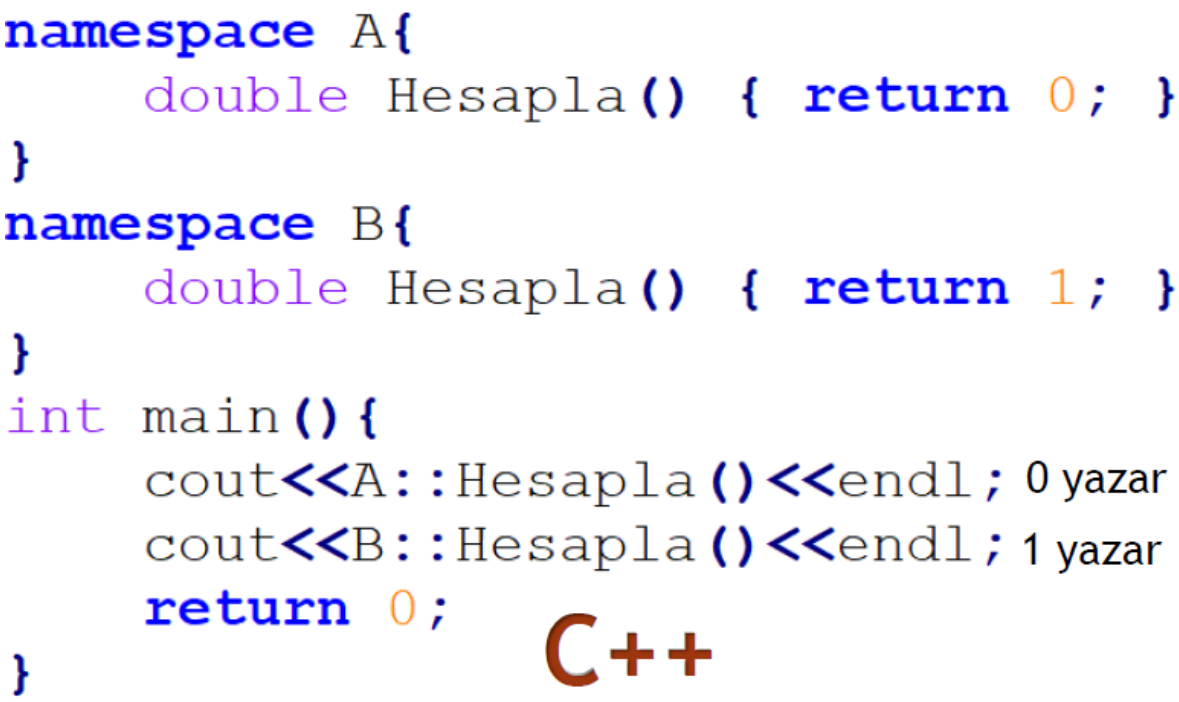
Python

Kurucu çağırma işlemini ise kalıtım alırken yazılan sıralamaya göre yapar. Yani önce Insan sonra Calisan sınıfın kurucusu çağrılır. Bu durumda 2 tane base sınıf olduğu için 2 defa super() yazılması gerekir ki 2.de Calisan'ın kurucusu çağırılsın.

Namespace (İsim Uzayları) ve Paketler

Bir arayüzden farklı olarak isim uzayları kodu organize etmek, derli toplu bir halde bulunmasını sağlar C++ ve C# dillerinde desteklenir.
Java dilindeki karşılığı paketlerdir ve klasör mantığı ile saklanır.

Örneğin C++'da std de bir isim uzayıdır.



İsim uzayı kaynak kodun içerisinde tanımlanmadığı zaman bu şekilde `A::` kullanımı yapılması gerekir. Örneğin main öncesi `using namespace A` demiş olsaydık bu durumda direkt olarak Hesapla() çağırdığımız yerde A'nın Hesapla fonksiyonu çağırılırdı.

Soyut Sınıflar

Arayüz ile kalıtım arasında bir yapıdır.

İçerisinde tamamlanmamış alanlar içerebilen sınıflara denir.

Arayüzden farkı tanımlanmış yani gövdesi dolu alanlar da içerebilirler.

Tanımlanmamış alanlar kalıtım yolu ile tanımlanır.

Eksik tanım içerebildikleri için soyut sınıflardan nesne türetilemez.

C++

```cpp
class Canli{
    private:
        int yas;
    public:
        virtual void YemekYe() = 0;
        int Yas()const { return yas; }
};
class Kedi : public Canli{
    public:
        void YemekYe(){

        }
};
```

C#

```csharp
public abstract class Canli{

    public int yas { get; }

    public abstract void YemekYe();
}

public class Kedi : Canli
{

    public override void YemekYe()
    {

    }
}
```

C#'da eğer sınıf abstract olarak tanımlanmayıp içerisinde metot abstract olarak tanımlanırsa bu durumda C# hata verir, sınıfı abstract yapmayı zorunlu kılar. Abstract sınıfı kalıtım alan sınıf yeniden düzenleyeceği veya gövdesi boş olan fonksiyonu override ederek yeniden yazar.

Kalıtım Hiyerarşisi

Java ve C# için kim kimden kalıtım alabilir;

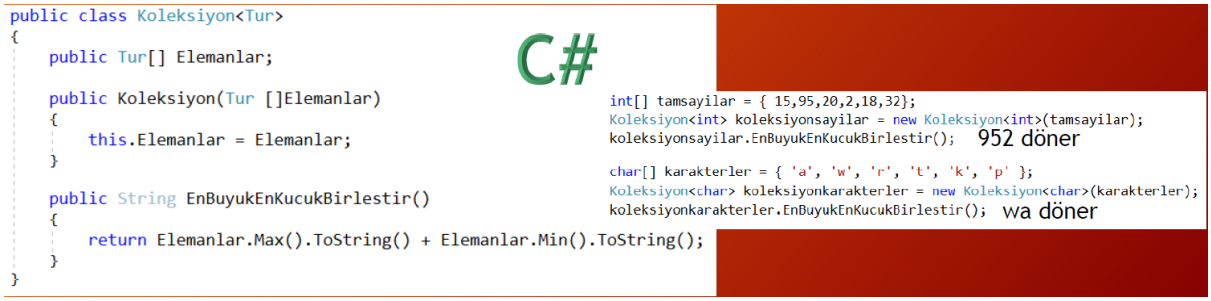
Object Veri Türü

Herhangi bir türü içinde barındırabilecek şekilde tasarlanmış veri türüdür.
Boxing ve Unboxing kullanımlarında büyük önem arz eder.
Java dilinde en üstteki sınıf Object sınıfıdır ve bu sınıftan diğer bütün sınıflar gizli olarak kalıtım alır.
Java ve C# dillerinde Object sınıfı bulunurken, C++ dilinde böyle bir sınıf yoktur.

C++ ve C'de void pointer'ı aynı işlemi görmek için kullanılabilir.

Generic Şablon Yapılar

C# ve C++ dillerinde aktif olarak kullanılan şablon yapılar Java diline sonradan dahil olmuştur.
Bir dilde Object sınıfı ile şablon yapıları taklit edilebilir.




```cpp
template <typename Tur>
class Koleksiyon{
    public:
        Tur *Elemanlar;
        int uzunluk;
        Koleksiyon(Tur *Elemanlar,int uzunluk){
            this->Elemanlar = Elemanlar;
            this->uzunluk = uzunluk;
        }
        string EnBuyukEnKucukBirlestir(){
            return toString(*max_element(Elemanlar,Elemanlar+uzunluk))+
                   toString(*min_element(Elemanlar,Elemanlar+uzunluk));
        }
        string toString(Tur t)
        {
            ostringstream ss;
            ss << t;
            return ss.str();
        }

};
```

C++

friend Erişim Niteleyicisi

Java dilinde friend diye bir terim yoktur fakat bu işlem sınıfları aynı pakete yerleştirerek kısmen gerçekleştirilebilir.
Aynı paketteki sınıflar birbirlerinin protected ve erişim niteleyicisi olmayan elemanlarına erişebilirler.

friend ibaresinin kullanıldığı yerde arkadaş olan sınıf o sınıfın private alanına erişebilir. Örneğin burada Sayi sınıfında Top sınıfı arkadaş sınıf tanımlaması yapılmış ve Top sınıfında getDeger metodunda s'nin değerine private olarak tanımlanmasına rağmen erişiliyor.

```
class Sayi{
    private:
        double deger;
        void setDeger(double dgr){
            deger = dgr;
        }
    friend class Top;
};
class Top{
    private:
        Sayi *s;
    public:
        Top(){
            s = new Sayi();
            s->setDeger(100);
        }
        double getDeger()const{
            return s->deger;
        }
};
```

****Fonksiyon sonuna const konulan fonksiyonlar, const üye fonksiyonu olarak adlandırılır. Bunların işlevi bu fonksiyon vasıtasıyla class içindeki değerlerin değiştirilemeyeceğini belirtir. (Read only amacıyla kullanılır)

*******C'de kalıtımın benzetimi nesne yönelimli → Hafta 9

C'de for içerisinde "i" tanımlaması yapılması standart değildir, tavsiye edilmez! Bu şekilde yapılması doğru olandır.

```
int i;
for(i=0;....)
```

Java'da 2 nesneyi if bloğunda == olarak karşılaştırırsak aslında burada bu 2 nesne aynı adresi mi gösteriyor diye sormuş oluruz. Örn;

```
Sayi s1=new Sayi(500);
Sayi s2=new Sayi(500);
if(s1==s2) System.out.println("Sayilar esit!");
else System.out.println("Sayilar esit degil!");
```

Burada normalde sayı1 ve sayı2 birbirine eşit fakat içerideki deger intleri birbirine eşit. Biz burada nesne karşılaştırması yaptığımızdan dolayı ve aynı adresi göstermedikleri için tabiiki eşit değil çıkacaktır.

Bu durum için == if kontrolü durumlarında Java'da equals metodu kullanılır. Bu metodu sınıf içerisinde override ederek(Aslen Object sınıfında bulunuyor) içeride düzenlemesini yaparız. Hatta bu metodu override sayesinde aynı sınıftan 2 nesneden başka olarak farklı sınıflardan da nesneleri yani elma ile armudu da karşılaştırabiliriz. Yani eğer Sayi sınıfı içerisinde şöyle bir override yaparsak;

```
@Override
public boolean equals(Object obj) {
  if(obj==null) return false;
  if(getClass()!=obj.getClass())  return false;
  final Sayi sy=(Sayi)obj;
  return this.deger==sy.deger;
}
```

→ getClass(), equals fonksiyonunu çağıran nesnenin sınıfını öğrenir.

Sonrasında main içerisinde şöyle eşitlik sorgularsak;

```
if(s1.equals(s2)) System.out.println("Sayilar esit!");
else System.out.println("Sayilar esit degil!");
```

Bu durumda "Sayilar esit!" çıktısı alabiliriz.

C'de bool bulunmamaktadır fakat typedef ve enumarate kullanımı ile benzetim yapılabilir.

```
typedef enum BOOL{false, true}bool;
```

enum tanımlaması "bool" adında bir değişken tanımlar ve bu bool değişken adının yalnızca false veya true olabileceğini ifade eder.

*******C'de soyut sınıfın nesne yönelimli → Hafta 10

*******C'de Java'daki Object örneği → Hafta 10

Java'da Object sınıfından `Object obj=new Daire();` şeklinde bir nesne türettiğimizi düşünürsek ve Daire sınıfının içerisinde yariCap değişkeninin public olarak tanımlandığını varsayarsak, `obj.yariCap` şeklinde erişmek istersen obj kendi türünün ne olduğunu bilmediğinden dolayı buna erişemez(boxing, unboxing). Fakat obj'e türünün Daire olduğunu söylersek erişebilir;

```
System.out.println(((Daire)obj).yariCap);
```

C'de nesne yönelimlide stringlerle uğraşılıyorsa, stringler muhakkak heap bellek bölgesinde oluşturulmalıdır.

Exception Handling

C++'da throw 200 gibi int vs. herhangi tipte bir hata fırlatılabilir fakat Java'da her şey nesne olduğundan dolayı yalnızca nesne fırlatılabilir.

Java'ya ait exception handling sınıf diyagramı;

Throwable bir arayüzdür. Yani bütün hata sınıfları Throwable arayüzünden kalıtım almak zorundadır.

Koyu olanlar yakalanması zorunlu hatalardır, açık renkliler ise opsiyonel.

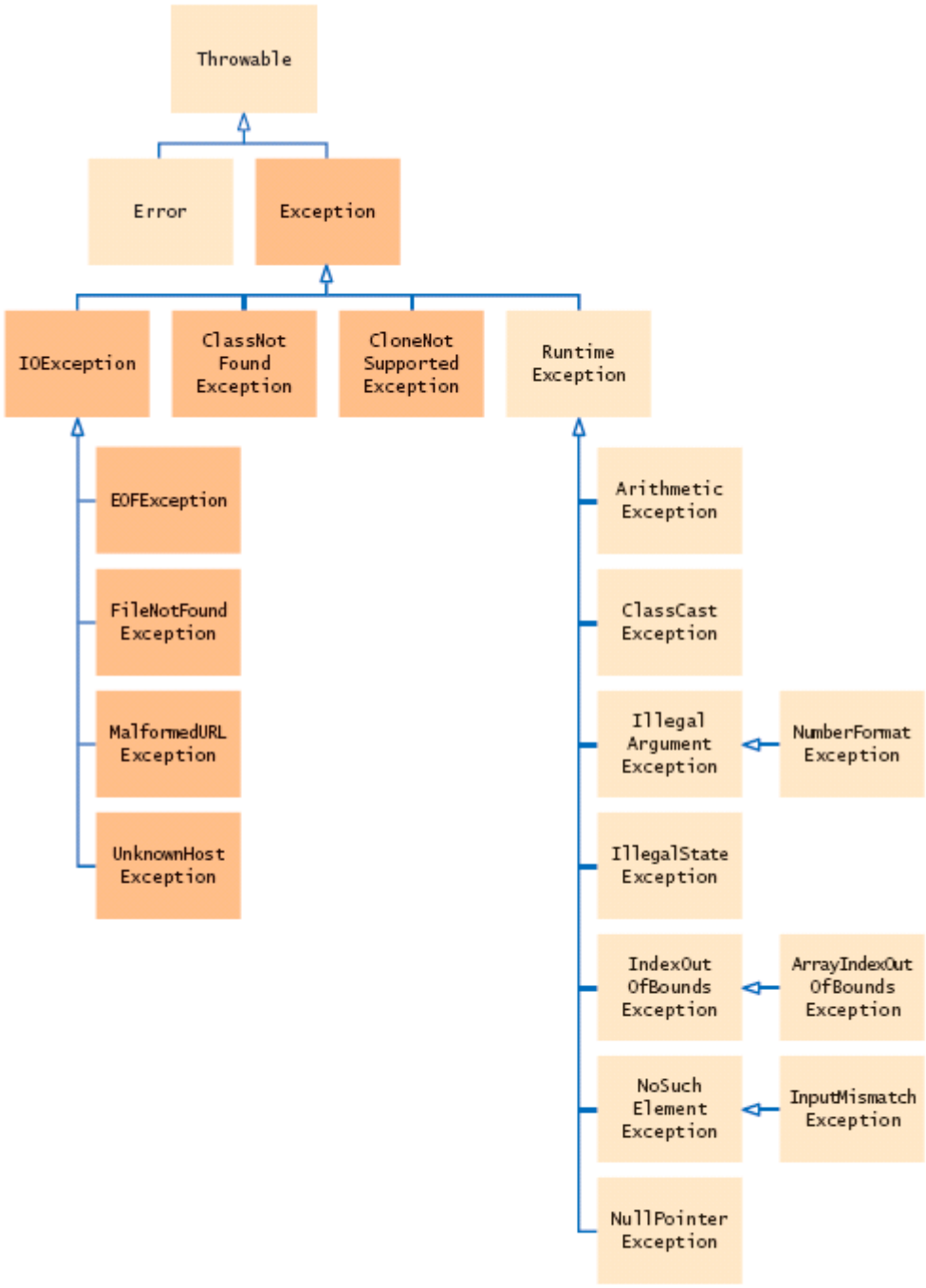


Figure 1 The Hierarchy of Exception Classes

Java örneği, diyelim ki banka hesabı sınıfımız var ve para çekmek için metot yazdık. Dolayısıyla çekeceğimiz para bakiyeden fazla olmamalı ve miktar bakiyeden büyük ise illegalargumentexception fırlat diyoruz ve parametre olarak da hata mesajının yetersiz bakiye olduğunu söylüyoruz.

```
public void ParaCek(double miktar) {
  if(miktar>bakiye) throw new IllegalArgumentException("Yetersiz bakiye!");
  bakiye-=miktar;
}
public void ParaYatir(double miktar) {
  if(miktar<=0) throw new ArithmeticException("Hatalı miktar!");
  bakiye+=miktar;
}
```

Sonrasında bu metodu çağıracakken eğer try catch yapmaz isek yani yalnızca;

```
Hesap hesap=new Hesap();
hesap.ParaCek(100);
```

bu durumda önce bakiye 0 olacağından kırmızı hata yazılarıyla programı durdurur ve illegalargument hatası fırlatır fakat biz bu para çekme metodunu try içerisinde çağırırsak ve beklediğimiz hatanın illegalargument olduğunu belirtirsek bu durumda eğer miktar bakiyeden fazla ise exception bloğunda istediğimiz işlemleri yapar. Örneğin burada "Yetersiz bakiye!" olarak girdiğimiz parametreyi yazdırmışız ve program beklenmedik bir şekilde sonlanmamış.

```
try {
  hesap.ParaCek(100);
} catch (IllegalArgumentException ex) {
  System.out.println(ex.getMessage());
}
```

Try bloğunda çok fazla sayıda satır kod olduğunu düşünürsek eğer, blok içerisinde kodlar işletilirken beklenen hata geldiğinde direkt olarak catch bloğuna atlar yani sonraki kodlar işletilmez. Yani örn bu örnekte hesap.ParaCek(100)'den sonra `System.out.println("SELAM");` kodu olduğunu düşünürsek bu durumda SELAM'ı print etmeyecektir çünkü ParaCek fonksiyonundan hata yakaladığından dolayı direkt catch bloğuna gidecektir.

Eğer try bloğundan farklı türlerde hatalar bekleniyorsa, VE her farklı hata türü için farklı işlemler yapılması planlanıyorsa her hata türü için farklı bir catch bloğu yazılabilir. Örneğin yukarıdaki örneğe ek olarak catch'in altına bir catch daha yazabiliriz.

```
try {
  hesap.ParaYatir(-1);
  hesap.ParaCek(100);
```

```
}
catch (ArithmeticException ex) {
  System.out.println(ex.getMessage());
}
catch (IllegalArgumentException ex) {
  System.out.println(ex.getMessage());
}
```

Eğer farklı hata türleri için farklı işlemler yapılmayacaksa beklenen yani throw edilmesi beklenen hata türleri 1 adet aynı catch bloğunda da "|" ile bulunabilir.

```
catch (ArithmeticException | IllegalArgumentException ex) {
  System.out.println(ex.getMessage());
}
```

Ya da tüm hataları kapsayan yalnızca Exception yazılabilir.

C'de ise throw, try-catch mekanizması bulunmamaktadır. Fakat bu işlemleri jumperlar yardımıyla yapabiliyoruz. Jumper'ları kullanabilmek için gerekli olan kütüphane `setjmp.h`'dır. Örneğin bir bölme işlemi yapıldığında eğer 0'a bölüm olursa bu beklenmeyen bir durum olur ve programın bozulmasına sebebiyet verebilir ve biz bu durumun önüne geçmek için global scopeda `jmp_buf` ile jumper adında bir jumper buffer tanımlaması yapıyoruz ve bunun değeri default olarak 0. Sonrasında bölme fonksiyonunu yazıyoruz ve aynı Java'daki mantık gibi eğer payda 0 ise `longjmp` metodu yardımıyla jumper'ın değerini -3 yap diyoruz. Main içerisinde ise try-catch'e benzer bir yapıyı if-else ile oluşturuyoruz. If şartında `setjmp` metodu yardımıyla jumper'ın değeri 0 ise yani default hali ise girip bolme fonksiyonunu çağırarak değeri yazdır diyoruz. Bolme fonksiyonu çağırıldığında eğer jumper'ın değeri değişip -3 olmaz ise yazdırıyor, -3 olursa yani y 0 ise o halde else bloğunu çalıştırarak sıfıra bölme hatası çıktısı veriyor.

```
#include "stdio.h"
#include "setjmp.h"
jmp_buf jumper;

int Bolme(int x, int y){
  if(y==0)  longjmp(jumper, -3);
  return x/y;
}

int main(){
  int a=10, b=0;
```

```
  if(setjmp(jumper)==0)
    printf("%d", Bolme(a,b));
  else
    printf("Sifira bolunme hatasi!");
  return 0;
}
```

Catch içerisinde de ters giden bir durumda hata fırlatılma olasılığı mevcuttur yani öngöremediğimiz bir durum olabilir. Try-Catch içerisinde her ne olursa olsun son olarak istediğimiz işlemleri her halükarda yapabileceğimiz finally bloğumuz da Java'da bulunur. Örneğin try'da hata alıp, catch'de de hata almamıza rağmen yani her koşulda son olarak hesabımızın bakiyesini yazdırmak isteyebiliriz.

Yani aslında bu durumda catch içerisinde alınan hata programı çökertecektir fakat program çökmeden önce finally bloğu bulunuyorsa oraya da giderek önce oradaki işlemleri yapacak.

```
try {
  hesap.ParaYatir(-1);
  hesap.ParaCek(100);
}
catch (ArithmeticException | IllegalArgumentException ex) {
  System.out.println(ex.getMessage());
}
finally {
  System.out.println(hesap);
}
-> Hatalı miktar!
-> 0.0
-> Exception in thread "main" java.lang.IllegalArgumentException: Yetersiz bakiye!
   at cc.Hesap.ParaCek(Hesap.java:13)
   at cc.Program.main(Program.java:15)
```

Bu şekilde bir çıktı üretir. Yani finally hata olsa da olmasa da çalışır. (Bu örnekte catch'te hatalık bir durum yok, örnek olması açısından çıktısı öyle yazıldı)

Eğer yazacağımız programda Java'nın sunmuş olduğu hata türleri bizim uygulamamızın akışında oluşabilecek hatalara uygun değilse yani istediğimiz hatayı göndüremiyorsak bu durumda kendi hata sınıfımızı kendimiz yazabiliriz.

Hata sınıfı Java'nın diyagramda bulunan herhangi bir hata sınıfından veya Throwable arayüzünden kalıtım alınarak sağlanabilir. Örn hesap örneği için

yetersiz bakiye hatası yazmak istersek;

```
public class YetersizBakiye extends IllegalArgumentException{
  public YetersizBakiye() {}
  public YetersizBakiye(String hataMesaji) {
    super(hataMesaji);
  }
}
```

Bu hata sınıfımızı yazdığımız durumda örneğin ParaCek fonksiyonunda IllegalArgumentException gibi şu şekilde YetersizBakiye hatası fırlatılabilir;

```
if(miktar>bakiye)  throw new YetersizBakiye("Yetersiz bakiye!");
```

Eş Zamanlı (Paralel) Programlama

Programın işlemci üzerinde paralel çalışmasıyla, programın kendisinin kod anlamında paralel çalışması farklı şeylerdir.

Bu zamana kadar görülen tüm anlatımlar, kodlar seri mantıkta yani tek thread ile çalışan kodlardı. Yani kaynak koddan 1. satırdan başlayarak 1 thread aşağı doğru son satıra kadar işleyişi çalıştırır. Fakat paralel programlamada aynı kod satırı içerisinden 1'den çok thread bulunarak o kod satırı 1'den fazla kez aynı anda çalıştırabilir. Yani genel anlamıyla paralel programlama 1'den fazla thread bulunan programlamadır. Yapay zekada paralellik hızlı sonuca ulaşılabilmesi adına çok önemlidir.

Mingw paralel programlamayı desteklememektedir dolayısıyla bu ders kapsamında yalnızca Java'da bu konu uygulamalı olarak işlenecektir.

Eş zamanlılığın amacı programın hesaplama hızını arttırıp verimi yükseltmektir. Bir programda eş zamanlılık kullanılmamışsa varsayılan olarak bir thread (ana thread denir) o programı yürütecektir.

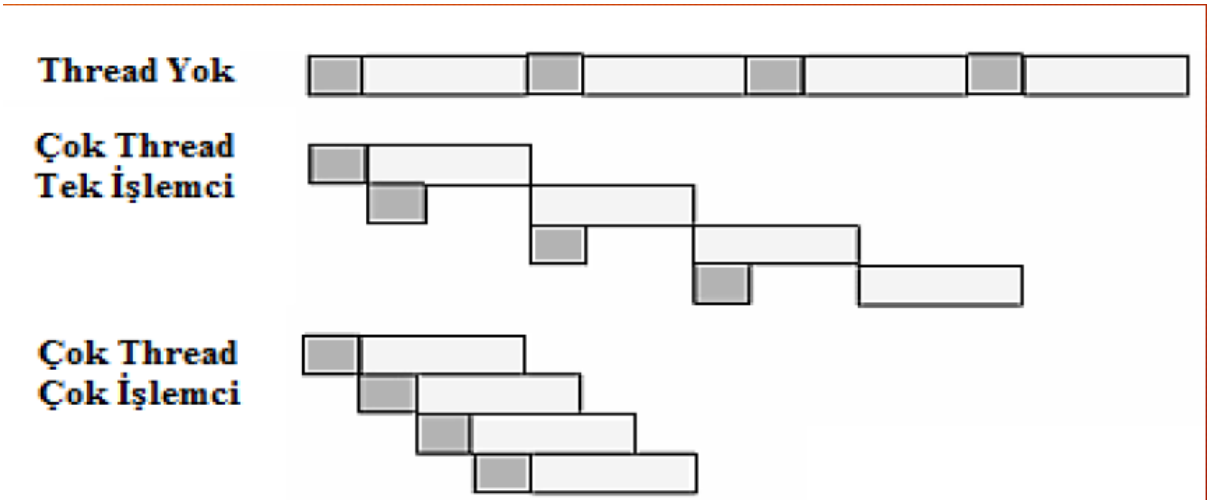
Bu konuda bir örnek vermek gerekirse bir duvarı 1 işçi 1 günde boyuyorsa 2 işçi yarım günde boyar. Fakat 100 işçi bu duvarı 1gün/100 sürede boyayamayacaktır çünkü bu 100 işçinin orada yaratacağı kalabalık vs. işin olmasına da engel olacağı için belki de 1 günden bile uzun sürecektir. Bundan dolayı gereksiz şekilde paralelleştirmeye gidilmemelidir. Yani küçük ölçekli işler için değil de büyük ölçekli işler için paralelleştirme mantıklıdır.

Mesela çok büyük boyutlu 2 matrisin çarpımını 1 thread'in çözmesi uzun sürecekken biz bunun çarpımı bölümlere ayırarak 4-5 thread ile bu threadlere farklı işler vererek çok daha kısa sürede çözülebilmesini sağlayabiliriz. Çünkü matris işlemlerinde birbirinden bağımsız bir çok hesaplama var ve bunları iş bölümlerine ayırarak rahatlıkla çok daha kısa sürede sonuca ulaşabilir hale getirebiliriz.

Örneğin matris çarpımında 100×100 gibi bir matrisi seri daha hızlı hesaplıyorken, paralel ise bu boyut büyüdüğünde 500×500 gibi bir matrisi serinin yarısı hızda 5 thread ile hesaplayabilir.

Bir paralel programlama yapacakken ana thread unutulmamalı yani bir işi 3 thread'e bölüyorum dediğimizde aslında biz bu programda 4 thread kullanıyor oluruz.

Donanimsal paralellik;



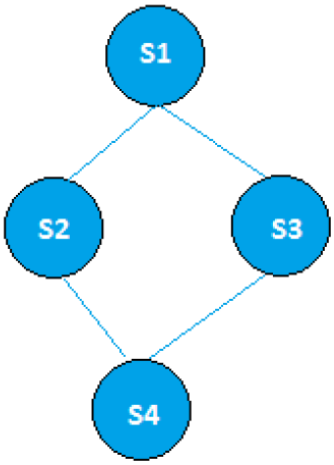
Program içerisindeki eş zamanlılık 4 farklı şekilde gerçekleşebilir.
Makine komutu düzeyinde: 2 veya daha fazla makine komutunun paralel çalıştırılması. Bu düzey kaynak kodun makine koduna dönüştürüldüğündeki yerde paralellik yapılmasıdır.
Program kod satırı düzeyinde: 2 veya daha fazla program kod satırının paralel çalıştırılması. Bu düzey bizim örnek yapacağımız kaynak kod üzerinde paralelliktir.
Birim seviyesinde: 2 veya daha fazla fonksiyonun paralel çalıştırılması. Bu düzey 2 veya daha fazla fonksiyonun paralel çalıştırılmasıdır.

Program seviyesinde: 2 veya daha fazla programın paralel çalıştırılması. Bu düzey 1'den çok programın paralel çalıştırılmasıdır. Fakat sonuç olarak 1 programın çalışmasına hizmet etmektedir. Örneğin Visual Studio çalıştıracakken bunun için arkada birçok başka ufak programın çalışması gibi.

Bu paralelliğin gerçekleştirilebilmesini bazı diller kütüphaneler yardımıyla gerçekleştirebilir.
Örnek: OpenMP C/C++ ve Fortran
Diğer bazı diller bunu kendi içerisinde sağlayabilir.
Bu şekilde ilk destek PL/I programlama dili ile başlamıştır. Daha sonra bunu, Ada95, Java, C#, Python ve Ruby takip etmiştir.

Bir programın hangi aşamada paralel, hangi aşamada seri çalışacağının belirlenmesi için öncelik grafları bulunur ve bu graflar hangi aşamada hangi işlemin başlayacağını belirtirler. Bir işlemin bitmesi ve bittikten sonra başlayacak olan işleme bağımlı işlem denir.

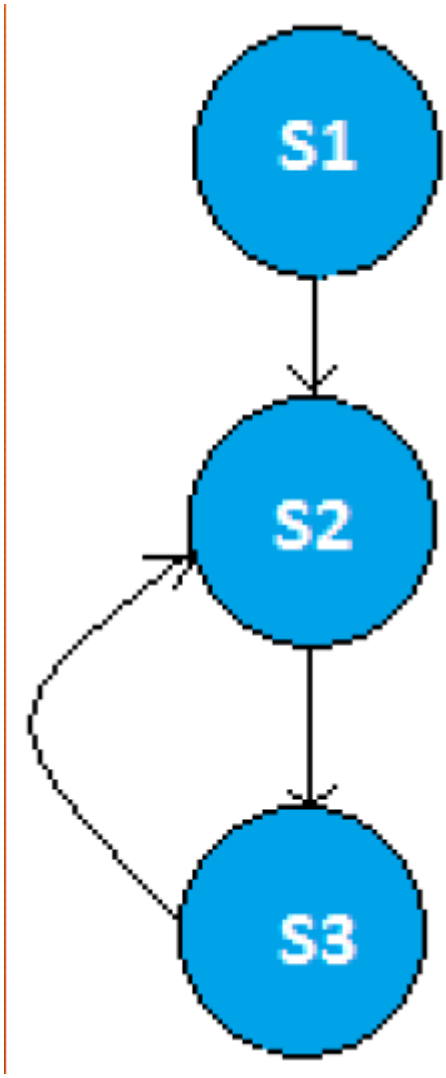
- S2 ve S3 çalıştırılabilmesi için S1 ifadesi işlemini bitirmelidir.
- S2 ve S3 eş zamanlı çalıştırılabilir.
- S4 çalışabilmesi için hem S2 hem de S3 işlemini bitirmelidir.



Yani bu örnekten şu ifade söylenebilir;

S1, (S2, S3) ve S4 ile seri çalışır.

Şöyle bir öncelik grafı hatalıdır çünkü S2'nin çalışması için S1 ve S3'ün bitmesini beklemektedir. Bu olay işletim sisteminde de sık olan bir olaydır ve buna deadlock denir.



Eşzamanlılık Şartları

Okuma ve yazma kütüphaneleri vardır. Eğer okuma ve yazma kütüphaneleri çıkartılabiliyorsa(her 2'si için de ayrı ayrı) o 2 işlemin paralel çalışabilip çalışamayacağına karar verebiliriz. Burada S1 5 satırlık, S2 10 satırlık bir kod da olabilir, fark etmez.

$R(Si) = \{a1, a2, \ldots, an\}$: Si için "oku" kümesi.

$W(Si) = \{b1, b2, \ldots, bn\}$: Si için "yaz" kümesi.

Paralel çalışabilmesi için 3 şart gereklidir;

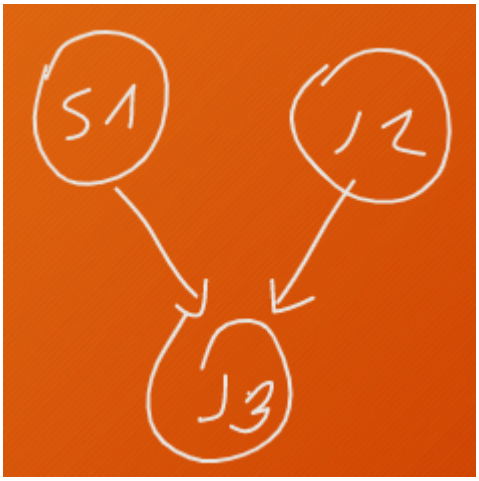
1. $R(S1) \cap W(S2) = \{\}$
2. $W(S1) \cap R(S2) = \{\}$
3. $W(S1) \cap W(S2) = \{\}$

Bu 3 işlemin sonucu da boş küme oluyorsa S1 ve S2 paralel çalışabilir demektir. Yani 1'i okurken veya yazarken diğeri okuyor veya yazıyor olmamalı, aynı şekilde 2'si de aynı anda yazmamalıdır.

Geniş kapsamlı 1 örnek;

*Okuma kümesinde değer sabitleri alınmaz(bkz S2).

Bu örneğin grafı ise şöyle olur;



Yani S1 ve S2 paralel çalışabiliyor fakat S3 çalışmak için S1 ve S2'yi beklemeli.

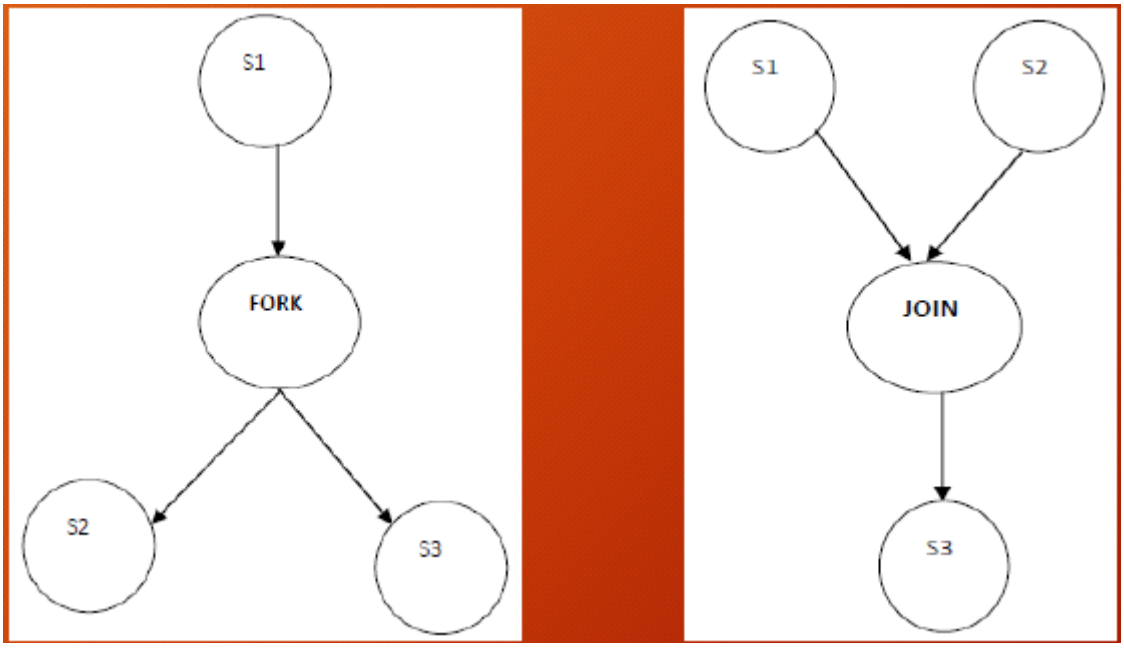
Fork ve Join Yapıları

Bu terimler eş zamanlılığı tanımlayan ilk programlama dili notasyonlarındandır.

Fork → gelen ana threadi fork kısmında bölünmesini sağlar. Yani paralelliğin başladığı nokta forktur.

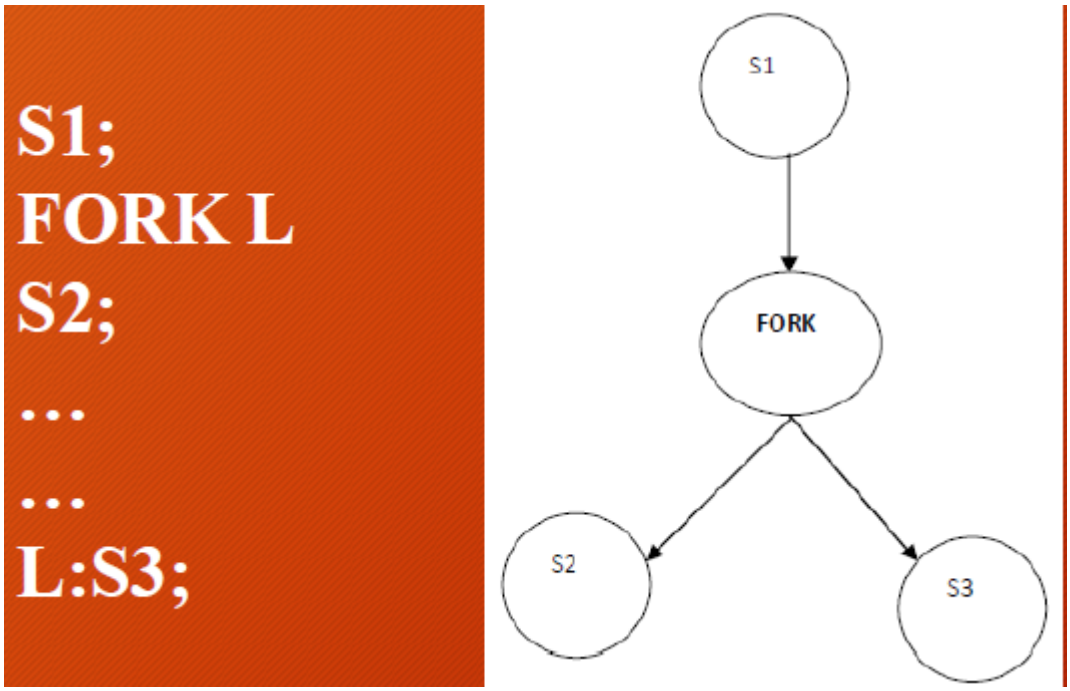
Join → bölünen threadlerin tekrar ana threadde toplanabilmesi için birleştirilmeleri işlemini yapar. Yani paralelliğin bittiği nokta joindir.

Join işleminde bir parametre beklenmektedir ve bu parametre kaç tane thread geleceğinin sayısıdır. Join bunu bilmek zorundadır çünkü paralel threadler farklı zamanlarda veya aynı anda işlerini bitirebilirler ve join bunu bilmezse ilk işini bitirip gelen threadde birleştirme yapacağını sanabilir.



Verilen koddan öncelik grafı veya verilen öncelik grafından kod çıkarma önemlidir!

Fork için örnek;

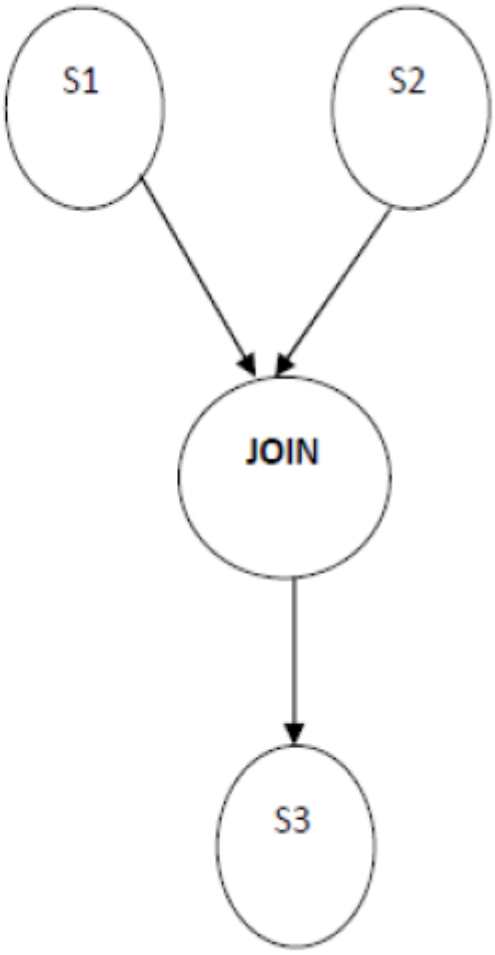


Örnekte direkt olarak S1; ifadesi seri olduğu anlamına gelir. Sonrasında FORK L ifadesinde L bir etiket ve bu etiket diğer dalın nereden devam ettiğini söylemektedir.

S'ler burada bir fonksiyon da olabilir, 3-5 satır kod da olabilir veya koca bir program da olabilir.

Join için örnek;

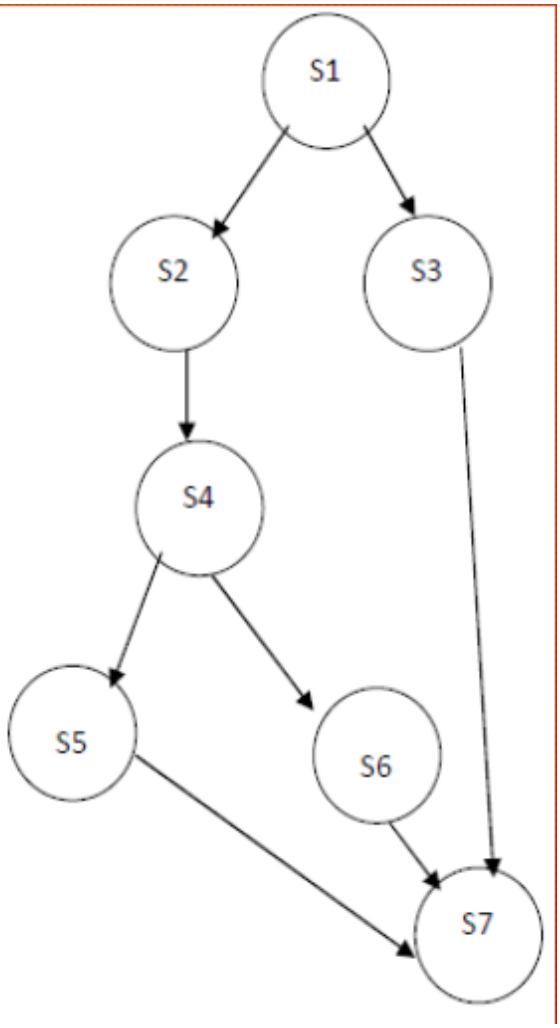
```
Count:=2;
FORK L1;
...
...
S1;
Go to L2;

L1:S2;
L2:JOIN count;
S3;
```



Örnekte L1:S2;'de diğer işlemler devam ederken S2 çalışmaya başlıyor ve aslında o sırada FORK L1;'den gotoya kadar olan kodlar çalışıyor durumda. S2 çalışınca L2'ye geliyor ve JOIN işlemi gerçekleşiyor fakat count başta 2 verildi, şu an 1 oldu ve bekliyor. Bu sırada yukarıdan devam eden işlemlerde S1 çalışıyor ve sonrasında goto L2; ile L2'ye gelip JOIN işlemiyle count 2 oluyor ve JOIN tamamlandını anlayıp birleştiriyor.

Fork ve join aynı kodda örnek; (Sınavda gelir)

```
S1
Count:=3;
FORK L1;
S2;
S4;
FORK L2;
S5;
Goto L3;
L2:S6;
Goto L3;
L1:S3;
L3: JOIN count;
S7;
```



Öncelikle S1'i görüyoruz ve S1'i çizdik, sonrasında FORK L1; demiş ve FORK L2'ye kadar hiçbir JOIN veya FORK ifadesi olmadığı için ve S2 ile S4 peşpeşe geldiği için demek ki S2 ve S4 seri çalışacaktır. S2 ve S4'ü seri çizdik. Daha sonrasında FORK L1 demiştik, L1'e gidiyoruz, S3'ü çiziyoruz. Sonrasında S4 sonrası FORK L2 ifadesi gelmiş, GOTO görene kadar yalnızca S5 var, S5'i çizdim ve L2'ye gittim. L2'de S6'yı gördüm ve S6'yı da çizdim. S5 peşinden şimdi Goto L3 gördüm ve L3'e gittim, JOIN diyor yani S5 birleşmeye gidiyor. Sonrasında L2 Forkundaki S6 ve sonrasında da L1 Forkundaki S3 de JOIN görüyor ve count 3 olup sondaki S7'ye birleşiyorlar.

Bu grafa göre S3'ün S2, S4, S5 ve S6'ya paralel çalıştığı söylenir.

Parbegin ve Parend Yapıları

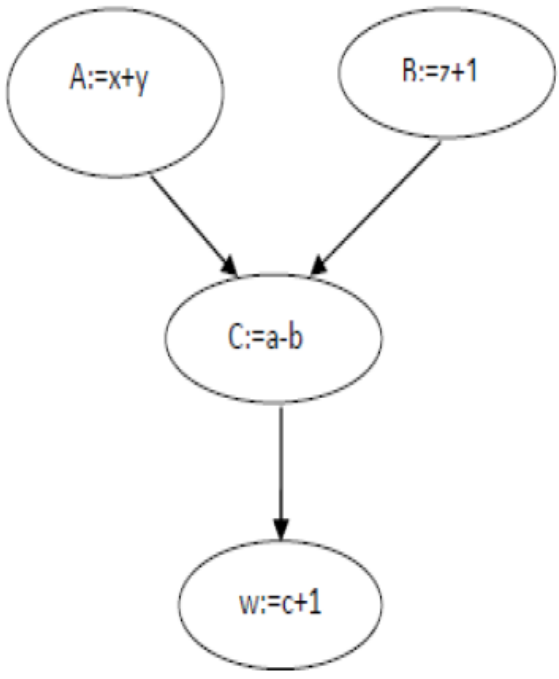
Fork ve Join gibi paralelliği belirten notasyonlardır. Bu yapıda Parbegin ve Parend'in arasında yazılan her şey paralel çalışır, farkı budur. Fork ve Join'e göre

anlaşılması daha kolaydır fakat bu yapıda ifade edilemeyecek bazı paralel yapılar vardır ve bunları ancak fork, join ile ifade edebiliriz.

Yani parbegin ve parend'in ifade ettiği bütün öncelik graflarını fork-join de ifade eder fakat fork-joinin ifade ettiği bütün öncelik graflarını parbegin-parend ifade edemez!! Fork-Join daha güçlü bir yapıdır.

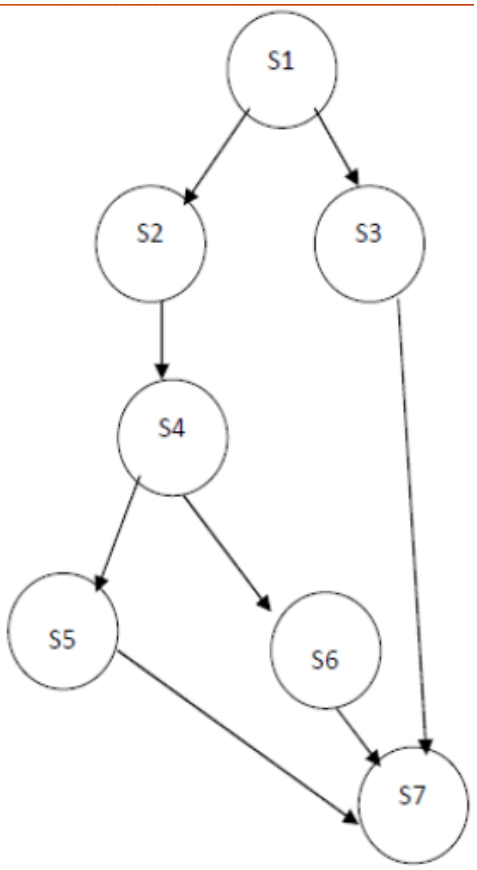
Ornek 1;

```
Parbegin
a:= x + y;
b:= z + 1;
parend;
c:= a – b;
w:=c + 1;
```



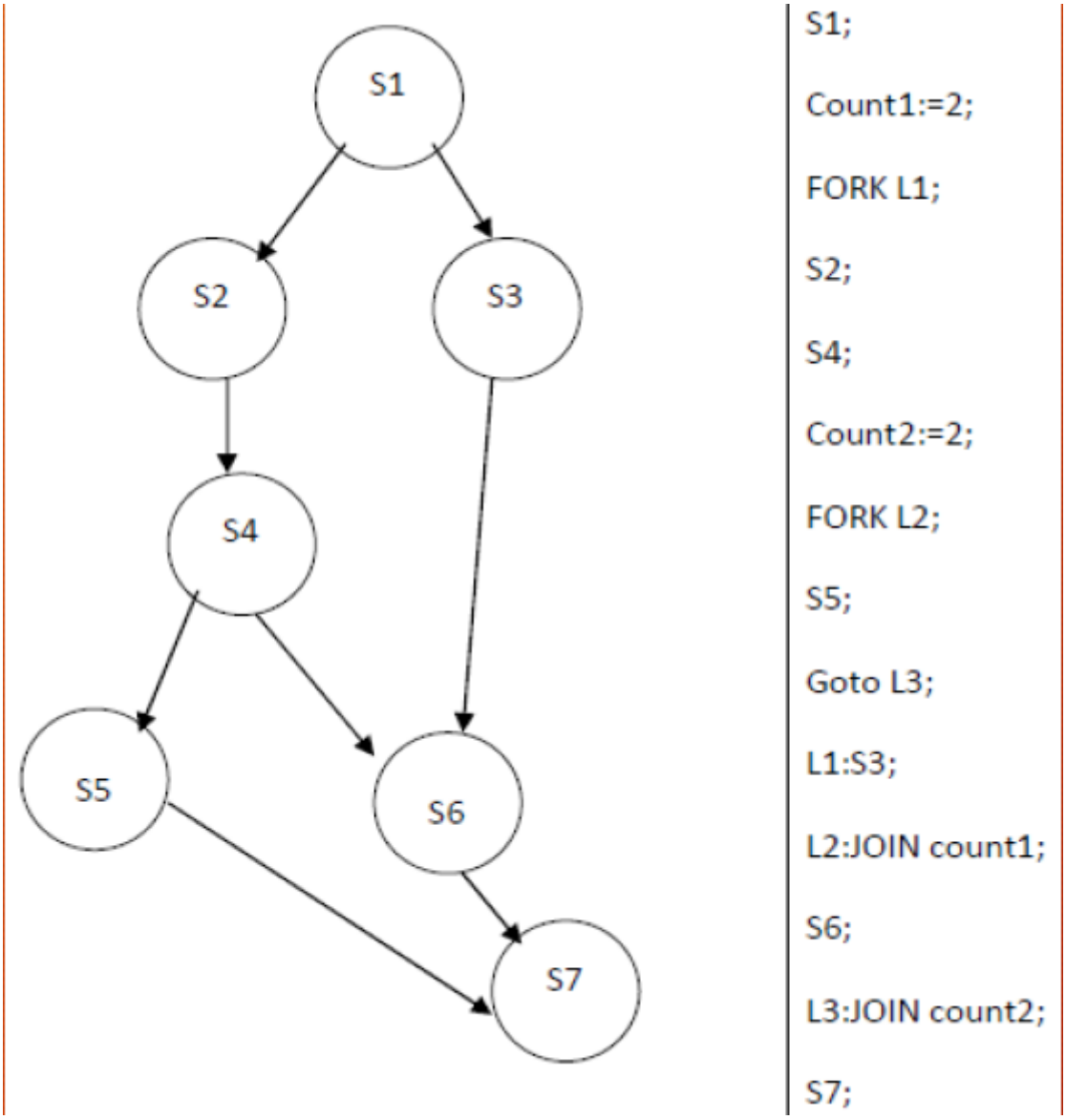
Ornek 2, fork-joinle yapılan örnek;

```
S1;
Parbegin
S3;
Begin
S2;
S4;
Parbegin
S5;
S6;
Parend;
End;
Parend;
S7;
```



Begin ve End ayrı bir blok olarak işleme tabi tutulur. Yani öncelikle S1'i gördü ve ardından parbegin başladı, S3 dedi ve ardından begin ile end olan kısmı ayrı tuttuk, begin içerisinde s2 ve s4 peşpeşe, seri çalıştırdık ve begin-end içerisinde parbegin-parend gördük yani s4 sonrası s5-s6 paralel çalışacak ve son olarak begin bitti, parend gördük ve s7'ye bağladık.

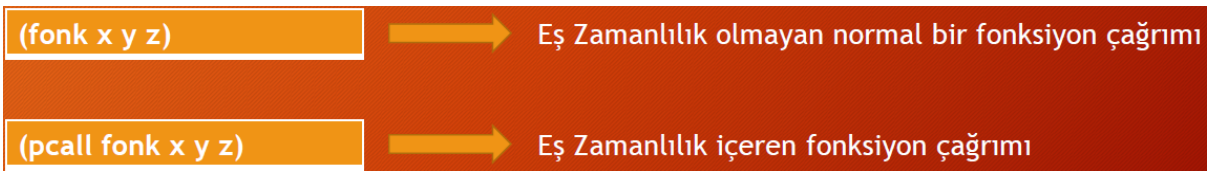
Örneğin böyle bir begin-end bloğunun içerisindeki paralelliğin içerisine bir gönderim yapmak parbegin-parendde mümkün değildir. Fork-Join ile mümkün, örn;



Bu örnekte L2 forku önce JOIN count1 daha sonra S6 çünkü önce S3'ün bitmiş olması gerekiyor ki S6'yı da forklayabilsin.

Fonksiyonel Dillerde Eş Zamanlılık

Multi-LISP fonksiyonel bir dil olmasına rağmen pcall yapısı ile paralel çalışmaya izin veriyordu.



pcall ile çağrılması x, y ve z parametrelerinin eş zamanlı çalışmasını sağlayacaktır. x, y ve z parametreleri yine bir fonksiyon olabilir bu durumda bu fonksiyonlar eş zamanlı çalıştırılır.

Thread Yield Metodu

Java'da yield metodu geçici olarak diğer threadlere zaman, yer verir. Yani sırasını başka threadlere verir ve geçici olarak yer verdiği threadler kullanılır. `Thread.yield();` şeklinde kullanılır ve o satıra gelen her thread bu komutu uygulayacaktır.

Thread Sleep Metodu

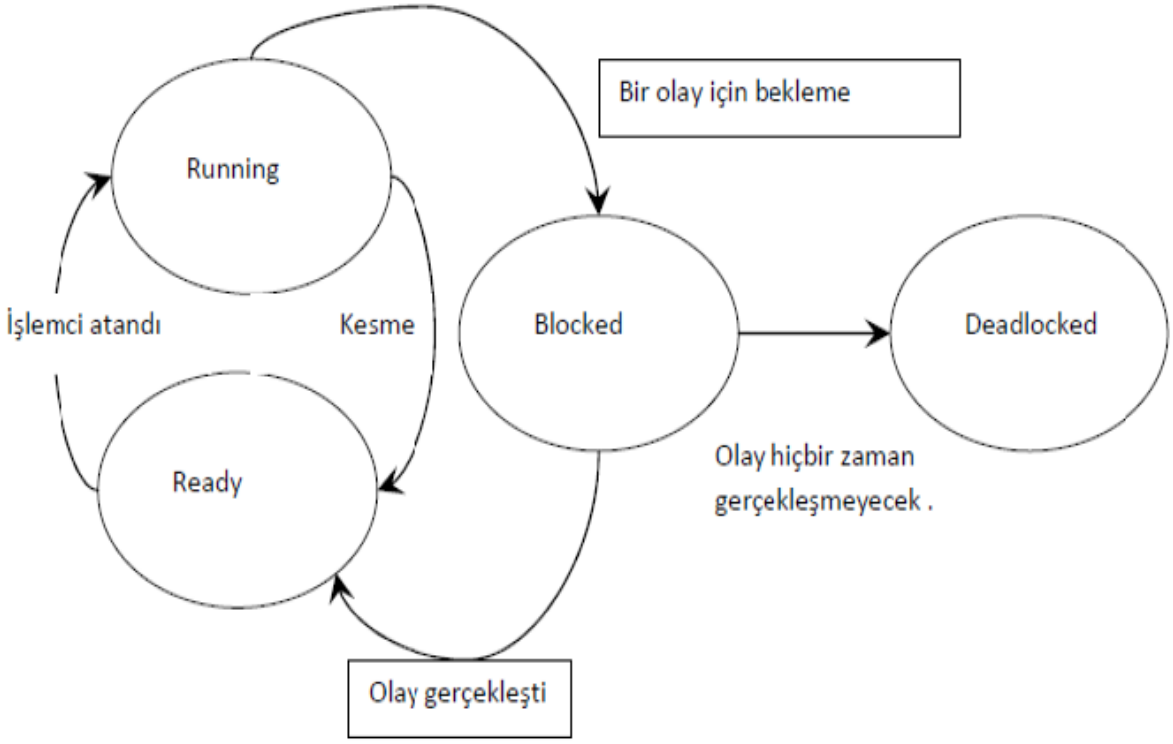
Belirtilen süre boyunca thread uykuya geçer. Bu metot tepki süresinin uzun olduğu bazı durumlarda zorunlu olarak kullanılabilmektedir.
`Thread.sleep(1);` şeklinde kullanılır. Örneğin bu kod threadi 1 milisaniye uykuya geçirir.
Java dilinde sleep metodu yakalanması zorunlu bir hata fırlatma durumu olduğu için try catch bloklarında kullanımı zorunludur. Yakalayacağı hata ise InterruptedException'dır. Deadlock veya ani beklenmeyen durumda thread'i kesmesi gibi durumlar için gereklidir.

**Sleep ve yield metotlarının kullanımı için eş zamanlılık şart değildir çünkü ana thread de bu metotları uygulayabilir.

İşlem Durumları

Bu durumlar thread'in işlemci üzerinde karşılaştığı durumlardır.

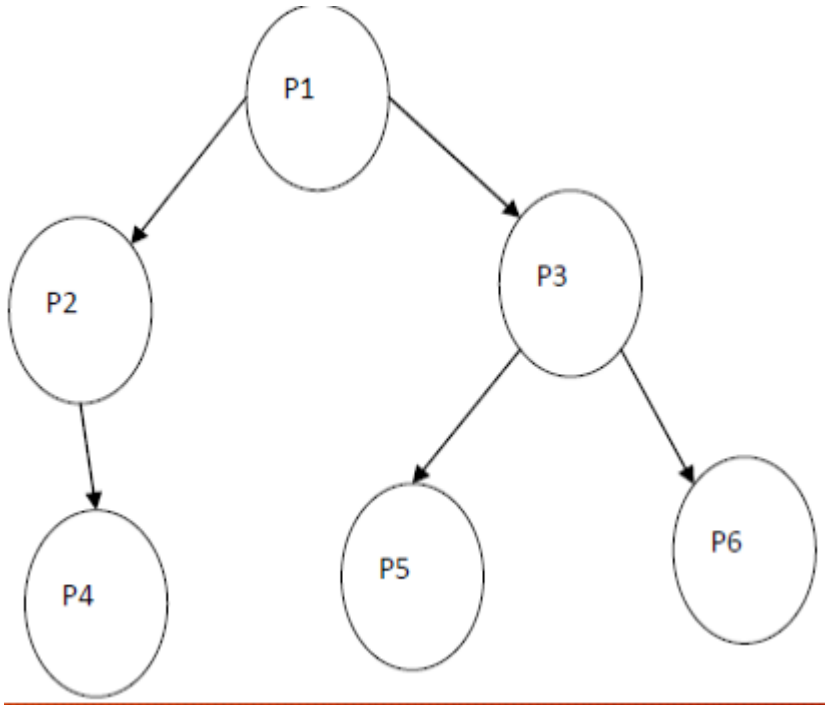
Running: Komutlar işletiliyor.
Blocked: Sistem bazı durumlar için bekletiliyor. Örneğin bir işlemin sonucunu bekleyebilir.
Ready: İşlem bir işlemciye atanmak için hazır durumda bekletiliyor.
Deadlock: İşlem hiçbir zaman gerçekleşmeyecek olayları bekliyor.



İşlem Grafları

Öncelik grafı ile farklı şeylerdir. İşlem grafı hangi işlem hangi işlemi oluşturuyoru söyler.

Örneğin;



Pi düğümünden Pj düğümüne gelen ok işareti Pi'nin Pj'yi oluşturduğunu ifade eder. Paralellikle ilgili hiçbir şey ifade etmez. Yani P3, P5 ile P6'yı oluşturmuş fakat bu P5 ve P6 paralel çalışıyor demek değildir!

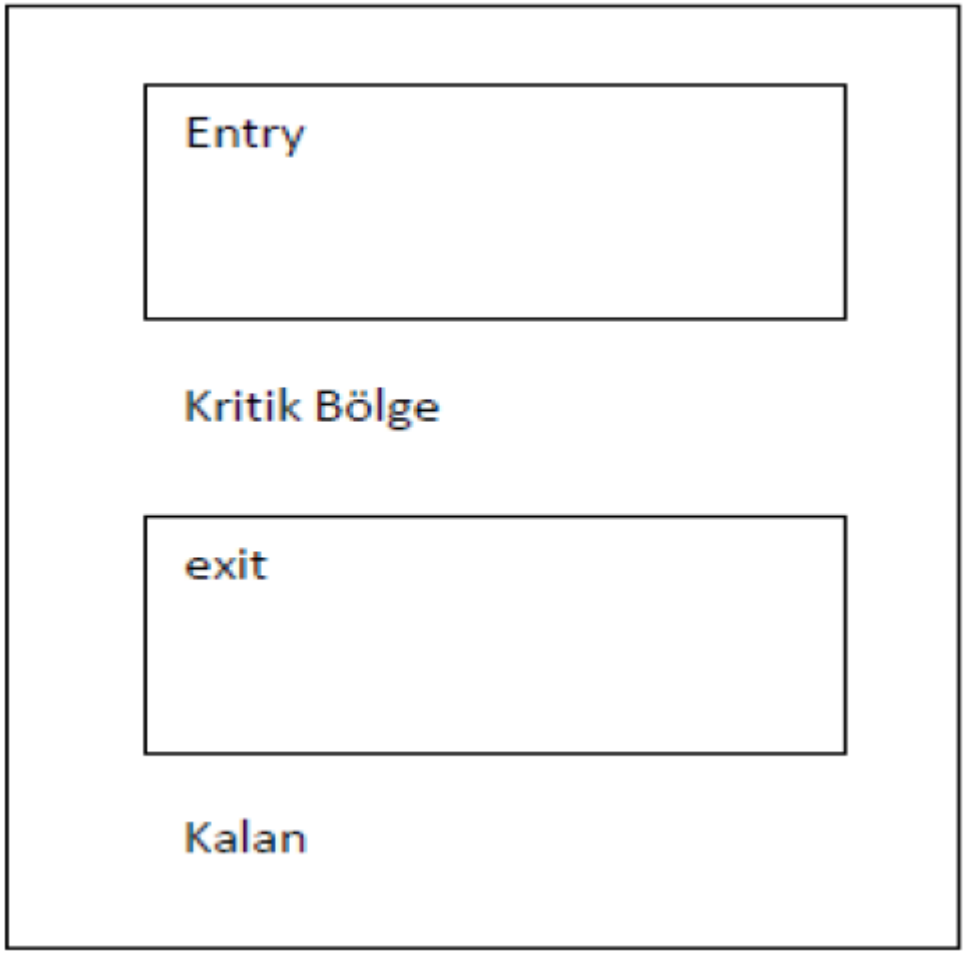
Kritik Bölge

Birlikte çalışan n adet işlemden {P1, P2, ..., Pn} oluşan bir sistem olduğu düşünülürse her bir işlem ortak değişkenleri okuyan bir tabloyu güncelleyen, bir dosyayı yazan vb. işlemleri içerebilir. Bu durumda seri programlamaya mahkum oluruz fakat kritik bölge tanımlayarak aslında bunu aşabiliriz.
Bu problemi aşabildiğimiz bu bölümlere kritik bölge ismi verilir ve bu bölgelere aynı anda sadece bir thread girebilir.

Yani aslında programımız paralel çalışsa çok daha hızlı çalışacakken eşzamanlılık şartlarında belirtilen 3 şartın da boş küme olmamasından kaynaklı yani paralel çalıştırılamayacağı kaynaklı bir durumda biz bu durumlarda paralelleştirilemeyecek bölgeyi kritik bölge oluştururarak problemi aşıyoruz.

Kritik bölge için YAZMA kümesi, yazma satırı kritik bölgeye alınır.

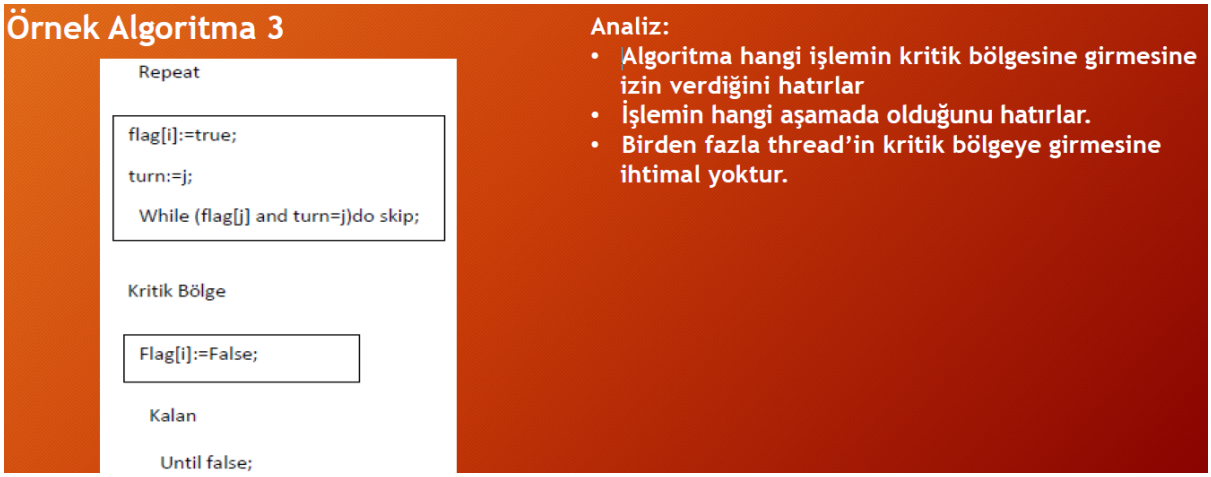
Yapısı şu şekildedir;



C# dilinde Parallel.For(){} şeklinde bir yapı vardır ve bildiğimiz, kullandığımız for döngüsünü paralelleştirir. Yalnız burada dikkat edilmesi gereken şey döngü içerisinde aynı noktaya yazma işlemi mevcutsa onun kritik bölgeye alınması gerekir, kritik bölgeye alma işlemini C# yapmaz fakat basitçe bizim için paralelleştirme işlemini yapar.

Kritik Bölge Gerçekleştirimi için Yaklaşımlar

Kütüphanelerin arkasındaki kritik bölge yapılarında bunun gibi gelişmiş yapılar vardır.



Semaforlar

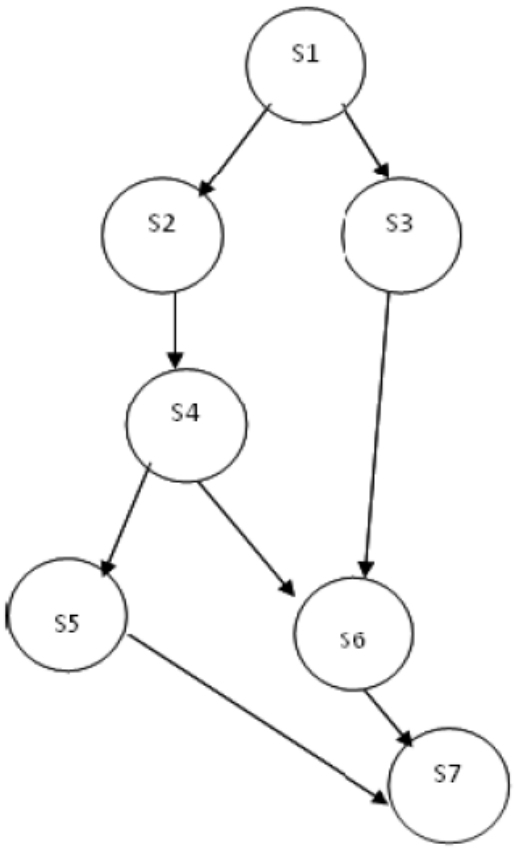
Karşılıklı hariç bırakma (mutual exclusuion) problemi için yapılan çözümleri daha komplex problemler için genelleştirmek kolay değildir fakat bu zorluğun üstesinden gelebilmek için semaforlar olarak adlandırılan bir senkronizasyon aracı kullanılabilir.

Yani paralel programlamadaki senkronizasyon problemini semaforlar yardımıyla aşabiliyoruz.

Semaforlar çok daha komplex kritik bölge problemlerini çözmek için kullanılırlar.

Semaforlarda P ve V değişkenleri vardır.(mutexler vs, bilinmesine gerek yok)

Parbegin-Parend ile yapılmayan fakat Fork-Join ile yapılabilen örneğin semaforlarla çözümü;




```
Var a, b, c, d, e, f, g: semaphore;
Begin
Parbegin
Begin S1; V(a); V(b) end;
Begin P(a); S2; S4; V(c); V(d); end;
Begin P(b); S3; V(e); end;
Begin P( c); S5; V(f); end;
Begin P(d); P(e); S6; V(g); end;
Begin P(f); P(g); S7; end
Parend;
End;
```

Başta Var tipinde tanımlanan a,b,c,d,e,f,g değişkenleri semaforlardır.

Begin S1 ile başlayan satır a ve b semaforlarının yani V olanlar P'lerin paralel çalıştığını söylüyor. Örneğin S1'de P(a) ve P(b)'nin S1'den paralel dallandığını söylemiş. P(a)'da S2 ve S4 seri çalıştığını ve paralel dalı olduğunu söylüyor, P(b)'de ise S3'ün paralel dalı olduğunu söylüyor. Daha sonrasında P(d)'de yani S6'ün forklanması için P(e) yani S3'ün bitmiş olması gerektiğini söylüyor.

Java'da bir thread'e görev verebilmek için işlem tanımlamak gerekir. Bunu yapmak içinse bir sınıfın işlem sınıfı olabilmesi için Runnable arayüzünden kalıtım almalıdır.

Bir işimiz var ve bu işi işçilere atamamız gerekiyor, thread burada işçi oluyor.

Örneğin parametre olarak verilen bir karakteri verilen adet kadar yazdıran ve parametre olarak verilen sayıya kadar sayıları yazdıran bir sınıf tasarlarsak;

```
public class SayiIslem implements Runnable{
  private int sonSayi;
  public SayiIslem(int s) {
```

```
    sonSayi=s;
  }
  @Override
  public void run() {
    for(int i=1; i<=sonSayi; i++)
      System.out.print(i+" ");
  }
}
public class KarakterIslem implements Runnable {
  private char yazilanKarakter;
  private int kacKere;
  public KarakterIslem(char c, int k) {
    yazilanKarakter=c;
    kacKere=k;
  }
  @Override
  public void run() {
    for(int i=0; i<kacKere; i++)
      System.out.print(yazilanKarakter);
  }
}
public class Program {
  public static void main(String[] args) {
    Runnable aYaz=new KarakterIslem('a', 100);
    Runnable bYaz=new KarakterIslem('b', 100);
    Runnable yaz100=new SayiIslem(100);

    Thread thread1=new Thread(aYaz);
    Thread thread2=new Thread(bYaz);
    Thread thread3=new Thread(yaz100);

    thread1.start();
    thread2.start();
    thread3.start();
    //yorum
    while(thread1.isAlive() || thread2.isAlive() || thread3.isAlive());
  }
}
```

Öncelikle Runnable arayüzünden oluşturduğumuz sınıfların gerçeklemesini yapıyoruz, nesne türetiyoruz ve ardından 3 adet thread oluşturup bu threadlere oluşturduğumuz işleri veriyoruz. Threadler işçi, runnable sınıflarımız iş oluyor. Threadleri oluşturduktan sonra thread'in start metodu ile threadi başlatabiliyoruz ve otomatik olarak sınıf içerisinde Runnable arayüzünden kodladığımız run metodu yani ne iş yapacağını bildiği metodu işliyor. Burada öncelikle thread1'in start edilmiş olması ilk olarak thread1 başlıyor demek olmuyor. Bu programın çıktısı işletim sisteminin o ana bağlı olarak farklı farklı çıktılar üretecektir. Yani düz baktığımızda her ne kadar önce thread1 başlatılıyor gibi gözükse aslında çıktıda önce thread2, thread2 bitmeden thread3 çıktıları vs görünebilir.

Biz threadlerin ne zaman işlemlerini bitirdiklerini bilemeyiz. Eğer sondaki while öncesi hiçbir şey yazmasaydık ve yorum satırında threadlerin düz mantıkla bitmiş olduğuna dayanarak bir işlem yapsaydık hatalı olurduk çünkü main bittiğinde threadlerin bitmiş olacağı garanti edilmez. Bundan dolayı while döngüsü ile threadlerin ölüp ölmediğini(işlerini bitirdiklerinde yok olurlar) kontrol ediyoruz, bu 3 thread'in bittiğinden emin olduktan sonra yapacağımız işlemleri ise bu while sonrasına yazabiliriz.

Yield kullanımı içinse sınıfların run metodunda `Thread.yield();` yazdığımızda uygulamış oluruz. Örn KarakterIslem için;

```
@Override
public void run() {
  for(int i=0; i<kacKere; i++){
    System.out.print(yazilanKarakter);
    Thread.yield();
  }
}
```

Sleep kullanımı içinse daha önceden belirtildiği gibi hata yakalanması zorunludur. Örneğin yine KarakterIslem için i'nin 50'den büyük olduğu durumlarda sleep metodunun çalıştırılması;

```
@Override
public void run() {
  for(int i=0; i<kacKere; i++) {
    System.out.print(yazilanKarakter);
    if(i>50) {
      try {
        Thread.sleep(100);
      }
      catch (InterruptedException e) {
      }
    }
  }
}
```

Bu örnekte bu if'e giren thread 100 milisaniye bekletilecektir fakat o beklerken tabii diğer threadler çalışmaya devam edebilir.

Bu sleep metodu main içerisinde ana thread'i dahi uyutmak için kullanılabilir.

Örnekteki şekilde her işlem için 1 thread pek de mantıklı değildir. Mantıklı olan 1 işi 1 iş havuzu oluşturarak belli sayıda threade vermek ve bu iş havuzlarıyla işi çözmektir. Yani örneğin 5 işçiyi a iş havuzuna verip bu işlem bittikten sonra diğer bir 5 işçi verilmiş b havuzundaki işi yaptırmak işin karmaşasını ve daha da uzayabilme ihtimalini düşürecektir.

Bu havuz işlemleri için Java'da ExecutorService kullanılır. Örn ilk örnekteki 3 işi 1 havuz oluşturup bu havuzda çalıştırmak istersek;

`ExecutorService havuz=Executors.newFixedThreadPool(3);`

burada parametre olarak kaç işlem vereceğimizi girdik. Bu komut aslında threadleri de bizim yerimize oluşturur. Yani verdiğimiz 3 parametresi kadar, 3 thread bu komutta oluşturulmuş oluyor. Şu an havuzumuzu oluşturmuş olduk ve artık bu havuza işlemlerimizi atmamız gerekiyor. Bu işlemi de execute metodu ile yapabiliyoruz. Burada normalde nesneyi arayüzden tek tek türetecekken parametre olarak direkt üretip verdik. Execute metodu aynı zamanda thread'i start ettiği yani başlattığı için start metodunu çağırmamıza gerek yok. Burada ek olarak ilk örnekte while ile threadlerin bittiğinden emin olmak için bir koşulla ana threadi bekletmiştik. Bu havuz yapısında bu işi threadlerin işi bittiğinde havuzun önce kendisini güvenli bir şekilde kapatması için shutdown metodunu kullanarak işinin bitmesini, sonrasında da havuz işini bitirene dek yeni satırlara atlayamasın diye while ile yok edildiğinin kontrolünü yapıyoruz. Shutdown işini bitirdiyse havuz yok edilmiş olacak. Eğer shutdown yapmasaydık bu sefer kendini işleri bitirse dahi terminate etmeyecekti. While döngüsüyle orada bekletmeseydik bu sefer de işleri bitirene dek alt satırları işlemeye devam edecekti.

```
ExecutorService havuz=Executors.newFixedThreadPool(3);
havuz.execute(new KarakterIslem('a', 100));
havuz.execute(new KarakterIslem('b', 100));
havuz.execute(new SayiIslem(100));
havuz.shutdown();
while(!havuz.isTerminated());
```

Burada parametre olarak 2 verip yine aynı şekilde havuza 3 iş execute etseydik bu defa 2 thread ile 3 işi yapacaktı. 1 verdiğimizde ise main ana thread mantığıyla teker teker sırayla tek thread ile işlemleri yapacaktı. Yani parametre 1 versek önce a'lar sonra b'ler sonra sayılar çıktı verecekti. Burada mühim olan 1 thread parametre verirsek o işi yapan 1 thread olacağıdır yani o 1 ana thread olmaz. Veya 100 parametre verip yani 100 thread ile bile bu 3 işi yaptırabilirdik

fakat normalde yapacağından çok çok daha uzun sürede bu işleri bitirecek olurdu.

Kritik bölge örneği yani paralelleştirme senkronizasyonu için tek bir yardım banka hesabımız olduğunu ve çok sayıda insanın bu hesaba aynı anda para gönderdiğini düşünürsek; bu para yatırma işlemi esnasında threadler yani işlemler çakışabilir. Teorik olarak küme olayında aynı yazma kümesine aynı işlemi yapmaya çalışıyor, aynı yere yazıyor olabilirler. Yani threadler aynı anda aynı işlemi yapmaya çalışabilir ve bu da işlemin gerçekleşmemesine sebep olur. Örneğin 100 adet 1TL hesaba yatırmayı 3 thread ile paralel denersek bunun başarı oranı %30 civarında olur. Yani hesapta 100TL beklerken 30'lu bir miktar görürüz. Bu sorunun örneği;

```
public class Hesap {
  private int bakiye;
  public Hesap() {
    bakiye=0;
  }
  public int getBakiye() {
    return bakiye;
  }
  public void paraYatir(int miktar) {
    int yeniBakiye=bakiye+miktar;
    //bu try bloğu sleep az thread ile denendiğinden denk gelme oranlarını arttırdık
    try {
      Thread.sleep(1);
    } catch (InterruptedException e) {}
    bakiye=yeniBakiye;
  }
}
public class ParaYatir implements Runnable{
  private Hesap hesap;
  private int miktar;
  public ParaYatir(Hesap h, int m) {
    hesap=h;
    miktar=m;
  }
  @Override
  public void run() {
    hesap.paraYatir(miktar);
  }
}
public class Program {
  public static void main(String[] args) {
    Hesap hesap=new Hesap();
    ExecutorService havuz=Executors.newFixedThreadPool(3);
    for(int i=0; i<100; i++) {
      havuz.execute(new ParaYatir(hesap, 1));
    }
    havuz.shutdown();
```

```
    while(!havuz.isTerminated());
    System.out.println(hesap.getBakiye());
  }
}
```

Bir hesap sınıfımız var ve banka hesabımızı tutuyor. Ardından ParaYatırma paralel işlemi için bir iş sınıfı oluşturduk. Son olarak ise main içerisinde bir 3 threadlik havuz oluşturup bu havuza 100 iş verdik. Sonuç bahsedilen sebeplerden dolayı normalde 100 çıkacakken 30 civarında bakiye çıkacaktır. Yani bu threadler bazı zamanlarda aynı yere yazmaya çalışıyorlar ve bu da paralelliği engelleyen bir kriter olduğundan dolayı burayı bir kritik bölge haline getirerek bu engeli aşmalıyız. Bu kritik bölge de o bölgeye yalnızca 1 thread girmesine izin vereceği için aynı anda giren thread sorununu aşarak istediğimiz sonucu alabileceğiz.

Bu işlemi de bu para yatırma işleminin gerçekleştiği metodu kritik bölge haline getirerek yapabiliriz. Hesap sınıfını şu şekilde düzenlersek;

```
public class Hesap {
  private int bakiye;
  private final Lock bolge;//+1
  public Hesap() {
    bolge=new ReentrantLock();//+2
    bakiye=0;
  }
  public void paraYatir(int miktar) {
    bolge.lock();//+3

    int yeniBakiye=bakiye+miktar;
    try {
      Thread.sleep(1);
    } catch (InterruptedException e) {}
    bakiye=yeniBakiye;

    bolge.unlock();//+4
  }
}
```

Öncelikle final yani sabit olarak Lock tipinde bolge adında bir değişken tanımladık ve bundan kurucuda nesneyi turettik. Daha sonrasında kritik bölge haline getireceğimiz metotta işlem öncesi `bolge.lock();` diyerek bölgeyi kilitledik ve içeride 1 thread var ise giriş, o çıkana dek kapalı halde kalacağı için diğer bir thread orada bekleyecek. Giren thread'in ise içerisinin tekrar boşaltıldığını bilebilmesi için unlock metodu ile metot sonunda açıyoruz. Bu işlem sonunda artık sonuç 100 veriyor oluyor.

Java'da programın belli bir kod aralığının çalışma süresi şu şekilde hesaplanabilir ve milisaniye cinsinden görüntülenebilir;

```
long baslangic=System.nanoTime();
..
..
..
long bitis=System.nanoTime();
double sure=(bitis-baslangic)/1000000;
System.out.println("Sure: "+String.format("%.2f", sure) + " milisaniye");
```

Fonksiyonel Programlama

Fonksiyonel programlama fonksiyon destekleyen anlamına gelmez çünkü son zamanlarda tüm programlama dilleri zaten fonksiyonları destekler, fonksiyonel programlama dili olan dilde her şey fonksiyonlardır. Örneğin bir fonksiyonun gövdesine indiğimizde orada yine fonksiyon vardır.

Fonksiyonel programlamanın en güçlü yanı liste işlemlerini yapabilmesidir.

Emir esaslı programlamada en önemli şey değişkenlerdir fakat fonksiyonel programlamada bu böyle değildir.

For, while olmadığından dolayı bu tür döngü durumlarında özyineleme kullanılır.

GNU Common Lisp hem derleyici hem yorumlayıcı olarak kullanılabilir. Masaüstünden kısayol olarak girdiğimiz ve konsol ekranında kodlar yazdığımız ekranda yorumlayıcı olarak çalışır. Bir dosyaya kodları yazıp sonrasında konsoluna girip onu çalıştırdığımızda ise derleyici olarak çalışır.

Genelde, en sık matematiksel işlemler için kullanılırlar.

Fonksiyonel olmayan tasarımlarda değişken(örnekte "x"), bir değeri tutan yer rolünü üstlenirken fonksiyonel tasarımda direkt değerin(örnekte "9") kendisidir.

`x = x + 1` ifadesinde her x farklı bir değeri temsil eder.
`10 = 9 + 1` deki gibi düşünülebilir.

Haskell Dili'nde;

Tembel değerlendirme (lazy evaluation) kullanır. (değer gerekmediği sürece hiçbir alt ifadeyi değerlendirme)

Liste kapsamları (list comprehensions) sonsuz listelerle çalışabilmeye izin verir.

Fonksiyonel programlama dillerinde liste dışındaki her şey atom olarak ifade edilir.

Lisp Dili Formları

ANSI Common Lisp (cLisp) → Derleyici, yorumlayıcı, debugger içerir.
GNU Common Lisp (gcl) → Derleyici, yorumlayıcı içerir.
Allegro CL (Commercial Common Lisp Implementation)

Lisp Dili Veri Türleri

İki ana veri türünden oluşur. → Atom ve Liste
Atom Veri Türü, String, Tam ve Ondalık sayılar ve Karmaşık sayılar içerir.

Fonksiyonel ile Emir Esaslı Tasarım Karşılaştırması

Emir Esaslı (imperative) diller	Fonksiyonel diller
Verimli çalışma	Verimsiz çalışma
Karmaşık semantik	Basit semantik
Karmaşık sentaks	Basit sentaks
Eş Zamanlılık (kullanıcı tanımlı)	Eş Zamanlılık (Otomatik)

Fonksiyonel dillerde eş zamanlılık otomatik olarak, gerekiyorsa ve yapılabiliyorsa program tarafından yapılmaktadır.

Lisp

Tüm komutlar parantezle başlar ve parantezle biter.

Lisp'de fonksiyonlar bir şey döndürmek zorundadır ve bunun için return vb ifadeler kullanılmaz, son yazdığımız satırdaki kodu döndürür. Yani ilk örnek için "Merhaba" öncesinde satırlarca kod olsaydı bile son satırda "Merhaba" olduğu için Merhaba'yı döndürürdü.

Lisp dilinde yorum satırı `;` ile başlar.

Bir fonksiyon tanımlamak için yine dile ait olan `Defun` fonksiyonuna parametre olarak fonksiyonumuzun adını ve alacaksa parametrelerini verip fonksiyon oluştururuz.

Lisp dilinde parametreler "," ile vs. değil, boşluklar ile ayrılır. Dolayısıyla Defun'dan sonraki ilk parametre yani boşluk sonrası ilk kelime bizim fonksiyon

adımız oluyor.

```
;PARAMETRESİZ
>(Defun Yaz()
"Merhaba"
)

;PARAMETRELİ
(Defun Hesap(x y)
;;;
)
```

Burada "Yaz" fonksiyon adı, hemen sonrasında gelen parantezler Yaz'ın parametre alanı ve en son Defun'ın kapatılacağı paranteze kadar olan kısımda kalan alan yani gövdesi de Merhaba yazan alan oluyor ve bu şekilde "Yaz" adında bir fonksiyon tanımlamış oluyoruz.

Lisp dilinde büyük-küçük harf duyarlılığı yoktur. Yazdığımız fonksiyonun adından örnek vermek gerekecek olursa bizim yazdığımız fonksiyonda her ne kadar adına "Yaz" da yazmış olsak, aslında büyük-küçük harf duyarsızdır.

Fonksiyon çağırmak içinse parantez içerisinde fonksiyonun adını yazmak ve varsa parametrelerini yazmak yeterlidir. (Parametresiz fonksiyonlarda boş parantezler yazılmaz!)

```
;PARAMETRESİZ
>(yaz)
"Merhaba"

;PARAMETRELİ (x ve y parametresi olsaydi)
>(yaz x y)
```

Derleyicili olarak kullanmak veya tüm fonksiyonları ayrı bir yerde yazıp tümünü tek seferde yorumlayıcı ekranına yüklemek istersek ".lisp" uzantılı dosya oluşturur, kodunu yazar ve öyle derleyebiliriz.

Lisp dilinde yazdırma işlemi `print` fonksiyonu ile yapılır.

`(print "x+y:")`

Lisp dilinde atama operatörü olmadığından bunu `setq` fonksiyonu ile halletmeye çalışırız. Örneğin burada setq sonuc adında ifade tanımlıyor ve Topla fonksiyonuna x ile y parametresini göndererek oradan dönecek ifadeyi sonuca atayarak sonucu elde etmiş oluyoruz.

`(setq sonuc (Topla x y))`

Lisp'de sayısal bir işlem yaptırmak istersek;

```
>(Defun Topla(x y)
  (+ x y)
)
```

şeklinde bir kodlama yaparız. Burada mühim olan x ve y'yi topluyoruz fakat görüleceği üzere başta "+" ifadesi girmişiz. Bu "+" ifadesi operatör değildir!! "+" ifadesi Lisp'in kendisinde bulunan bir fonksiyon adıdır! Aynı şekilde çıkarma için "-", çarpma için "*" ve bölme için "/" fonksiyonları da bulunur.

GNU Common Lisp konsolunda yazdığımız lisp dosyasındaki tüm fonksiyonları, tüm kodları tek seferde yüklemek istersek yine fonksiyon olan "load" ile bu işlemi dosyanın yolunu vererek yaparız.

`(load "D:/Users/Baris/Desktop/Program.lisp")`

Eğer dosya başarıyla yüklenirse `load` fonksiyonu bir bool fonksiyon yani doğru veya yanlış bir sonuç döndüren fonksiyon olduğu için T veya N döndürür. Lisp dilinde True veya False yoktur, True veya Nil vardır ve bunlar "T", "N" olarak ifade edilirler. Fakat Nil aynı zamanda NULL anlamındadır da.

```
Loading D:/Users/Baris/Desktop/Program.lisp
Finished loading D:/Users/Baris/Desktop/Program.lisp
T
```

Bu işlem başarıyla gerçekleştikten sonra kodun içerisindeki fonksiyonları çağırabiliriz. Yani load'un mantığı yazdığımız lisp kodundaki tüm fonksiyonları tek seferde tanımlamaktır.

GNU Common Lisp konsolunda yazdığımız lisp dosyasını derlemek istiyorsak yine bir fonksiyon olan `compile-file` kullanırız.

`(compile-file "D:/Users/Baris/Desktop/Program.lisp")`

Bu işlem sonucunda eğer bir hata ile karşılaşmazsak aynı konumda programın adında ".o" yani bu örnek için "Program.o" çalıştırılabilir dosyası oluşur.

Lisp dilinde kullanıcıdan bir veri almak için yine bir fonksiyon olarak "read" kullanılır ve setq ile de okunan değer bir ifadeye yazılır(x ve y). Örneğin

kullanıcıdan alınan 2 sayının kareleri toplamını ekrana yazdıran bir fonksiyon yazmak istersek;

```
(Defun Kare()
  (print "x:")
  (setq x (read))
  (print "y:")
  (setq y (read))
  (print "x^2+y^2=")
  (+ (* x x) (* y y))
)
```

Günümüzdeki dillerde sık sık bulunan "++" ve "--" operatörlerini de yine fonksiyonlarla Lisp dilinde kullanmak mümkündür. "incf" değeri 1 arttırır, "decf" ise değeri 1 azaltır.

`(incf x)`

`(decf x)`

Lisp dilinde 2 ifadenin değerlerini değiştirmek için `rotatef` fonksiyonu kullanılır.

`(rotatef x y)`

GNU Common Lisp'de varsayılan sayı tabanları 10'dur fakat farklı tabanlarda da sayı tanımlanabilir. Bu da diez `#` işareti ile ifade edilir. Tanımlamayı 2 tabanında yapmak için "#b" ile yaparız fakat değeri yazdırdığımızda bize yine 10 tabanında gösterir.

`(setq x #b001)`

Çıktısı 1 yani 10 tabanında gösterir.

Karmaşık sayıları destekleyen Lisp dilinde karmaşık sayı tanımlamak için yine # ile "#c" ile parametre olarak da gerçek ve sanal kısımlarını vererek karmaşık sayı oluşturulabilir.

`(setq x #c(1 5))`

Dolayısıyla 2 sanal sayıyı da + fonksiyonu ile toplayabiliriz veya başka işlemler yapabiliriz.

`(+ #c(12 4) #c(1 5))`

Lisp dilinde koşul ifadesi `if` fonksiyonu ile tanımlanabilir. İf fonksiyonu tanımlanmasında if'in ilk parametresi koşul, 2. parametresi eğer koşul true ise ne döndüreceği, 3. parametresi ise koşul nil ise ne döndüreceğidir. Aynı zamanda if fonksiyonu yazarken && yani ve veya || tanımlamaları da if'in 1. koşulunda and ve or ile ifade edilebilir. Verilen parametreye göre o yaşın genç bir yaş mı olduğunu söyleyen bir fonksiyon yazacak olursak;

```
(Defun Gencmi(yas)
  (if (and (> yas 10) (< yas 40)) T nil)
)
```

Görüleceği üzere and ile yaşın 10'dan büyük ve 40'dan küçük olduğu durumlarda T, aksi halde nil döndürmesini tanımlamışız.

Lisp dilinde for-while-until döngü tanımlaması ise `loop` fonksiyonu ile yapılabilir. Bu loop fonksiyonunu tanımlarken sonraki parametreye for girersek bunu sayaç olarak işleteceğimizi bildirmiş oluruz. Örnekte 1'den 10'a kadar i'yi print etmesini ifade etmişiz. Yukarıya doğru ilerlemesi yerine "downto" ile azaltarak da yazdırabilirdik.

```
(Defun Artan(sayi)
  (loop for i from 1 to sayi do (print i))
)
```

Bu çıktının en sonunda Nil yazdırır çünkü içerideki döngüden bir şey gelmez.

Yine döngü ile geriye doğru yalnızca tek sayıları ekrana yazdıran bir fonksiyon yazmak istersek;

```
(Defun AzalanTek(sayi)
  (loop for i from sayi downto 1 do
    (if (/= (mod i 2) 0) (print i))
  )
)
```

burada do'da yapılan işlem ise eşit değil mi operatörü olan `/=` ile `mod` fonksiyonundan i'nin 2'ye modu, sonraki parametredeki ifade yani 0 ise çift olduğu anlamına gelecek eşit değil mi'den dolayı ve sonraki parametrede bir önceki if örneğinde T olarak ifade ettiğimiz yani ilk parametrenin T olduğu yerde

i'yi yazdırıyor, Nil olduğu durum içinse hiçbir şey girmeyerek o durumlarda bir çıktı vermemesini sağlıyoruz.

Until ile faktöriyel hesaplayan bir fonksiyon yazmak istersek;

```
(Defun Faktoriyel(x)
  (setq sonuc 1)
  (loop until (< x 2) do
    (setq sonuc (* sonuc x))
    (decf x)
  )
  sonuc
)
```

Görüleceği üzere önce sonuc adlı bir değere 1 atanıyor ve loop fonksiyonu ile until tanımlanıp x 2'den küçük olana dek do'dan sonraki 2 ifadenin yapılması sağlanıyor ve en son loop bittiğinde ise sonuc'un return edilmesi ile faktöriyel hesabı yapılmış oluyor.

Lisp dili inanılmaz hızlı bir dildir. Örneğin bir önceki örnekteki faktöriyel örneğinde 10bin faktöriyeli dahi inanılmaz kısa sürelerde çözebilir. Bu hızından dolayı zamanında yapay zeka dili olarak tercih edilmiştir.

Özyineleme ile fibonacci hesaplayan bir fonksiyon yazmak istersek;

```
(Defun Fib(x)
  (if (< x 2) x
    (+ (Fib (- x 2)) (Fib (decf x)))
  )
)
```

Eğer x 2'den küçükse dolayısıyla direkt kendisini, küçük değilse 1 ve 2 eksiğinin fiblerinin toplamlarını yazdırıyor.

While ile fibonacci hesaplayan bir fonksiyon yazmak istersek;

```
(Defun Fib(x)
  (setq onceki 0)
  (setq simdiki 1)
```

```
  (setq i 1)
  (loop while (< i x) do
    (setq toplam (+ onceki simdiki))
    (setq onceki simdiki)
    (setq simdiki toplam)
    (incf i)
  )
  toplam
)
```

Bu fibonacci hesaplama fonksiyonu özyinelemeye göre çok çok daha hızlı bir şekilde hesaplama işlemini yapacaktır.

Lisp dilinde `list` fonksiyonu parametre olarak verilecek değerleri liste yapısına dönüştürür.

`(list x y z)`

Lisp dilinde eğer fonksiyonun parametrelerinden bazılarını opsiyonel olarak girmek veya girmemek istiyorsak bunu `&optional` ile sağlayabiliyoruz.

```
(Defun Fonk(x &optional y z)
  (list x y z)
)
```

Örneğin burada x'i girmek zorunlu fakat &optional yazdıktan sonraki parametreleri yani y ve z'yi girmek zorunlu değildir. Onları girmez isek onlara Nil yani null atar. Burada x'i ve y'yi girmek gibi bir şey yapabiliriz fakat x'i ve z'yi girmek gibi yani y'yi girmeden z'yi girmek gibi bir şeyi &optional ile sağlayamayız.

Böyle bir şey yapmak istersek bunu da `&key` ile sağlayabiliyoruz. Bu &key ifadesini de parametrelerin başına yazıyoruz. Bu durumda fonksiyonu çağırırken parametreleri verecekken önce parametrenin adını `:` ile belirtiriz. Fonksiyonun tanımlanması;

```
(Defun Fonk(&key x y z)
  (list x y z)
)
```

Bu fonksiyonu çağırıyorsak ve yalnızca x ile z'yi parametre olarak vereceksek;

`(Fonk :x 3 :z 5)`

Lisp dilinde 2 String ifadeyi birleştirmek istersek bunu `concatenate` fonksiyonu ile yapabiliriz.

```
(Defun Birlestir(isim)
  (setq nick (concatenate 'string isim "2021"))
  nick
)
```

Concatenate fonksiyonuna bir string birleştirmesi yapacağımızı `'string` ile belirterek sonrasında birleşecek stringleri veriyoruz ve bunu nick adında bir değişkene yolluyoruz. Fonksiyonu çağırırken vereceğimiz parametre string olmalı yani " ile başlayıp bitmelidir.

Lisp dilinde "*" ile başlayıp biten bir ifade, tanımlamanın bir liste olduğu anlamına gelir. Yani *A* şeklinde bir tanımlama yaparsak dil bunun bir liste olduğunu anlar. Bu listeyi tanımlamak içinse öncelikle `defparameter` fonksiyonu kullanılır. A adında 1 2 3 4 içeren bir liste tanımlamasak istersek;

`(defparameter *A* (list 1 2 3 4))`

Bu listeyi yazdırmak istersek yine "*"lar ile çağırmalıyız; `*A*`

Liste yapısına ait birden fazla fonksiyon mevcuttur. Bu fonksiyonlar kullanıldığında listenin orijinalinde herhangi bir değişiklik olmaz! Yani yaptığımız işlemi başka bir listeye atmazsak yalnızca print eder, atarsak atama yapar fakat kendinde değişiklik yapmaz. Buna `delete` fonksiyonu bile dahildir. (yalnızca push ve pop hariç)

Bu tanımladığımız listeye ait ilk elemana erişmek istersek;

`(first *A*)`

Bu tanımladığımız listeye ait ilk eleman hariç diğer elemanlarını listelemek istersek;

`(rest *A*)`

Bu tanımladığımız listeyi tersten listelemek istersek;

`(reverse *A*)`

Bu tanımladığımız listede bir eleman aramak istiyorsak; (eleman varsa elemanı (örnek için 5'i), yoksa nil döndürür)

`(find 5 *A*)`

Bu tanımladığımız listede parametre olarak verdiğimiz elemanın indeks numarasını istiyorsak; (eleman varsa elemanın indeksini (örnek için 0'ı), yoksa nil döndürür)

```
(position 1 *A*)
```

Bu tanımladığımız listeden bir eleman silmek istiyorsak;

```
(delete 1 *A*)
```

Eğer bu silme işleminin listeye etki etmesini istiyorsak kendi üzerine setq ile atama yapılabilir (tüm fonksiyonlar için geçerli);

```
(setq *A* (delete 1 *A*))
```

Tanımlanmış 2 listeyi birleştirmek istersek;

```
(append *A* *B*)
```

Tanımlanmış 2 listeyi karşılaştırmak istersek; (Elemanları değil, aynı listeyi yani aynı adresi gösterip göstermediklerini karşılaştırır)

```
(eq *A* *B*)
```

Bu 2 dizinin elemanlarını karşılaştırmak istersek bunun için bir fonksiyon yazarak bu işi yapabiliriz.

Bu tanımladığımız diziye yeni bir eleman eklemek istersek;

```
(push 9 *A*)
```

Bu fonksiyon yeni elemanı listenin sonuna değil, başına ekler.

Bu tanımladığımız diziden son elemanı çıkarmak istersek;

```
(pop *A*)
```

Bu pop fonksiyonu listeye son eklenen elemanı çıkarır. Yani push fonksiyonuyla A listemize en son 9'u eklemişsek ve push fonksiyonu kendi içerisinde aslında bunu listenin başına koymuş olsa dahi en son bu eleman eklendiği için pop'un çıkaracağı eleman yine 9 olur.

**push ve pop metotları listeye direkt etki eder. Yani önceki fonksiyonlardaki gibi geçici listeler oluşturup yeni listeyi yazdırmaz, direkt olarak orijinal listeyi değiştirir.

Burada ekstra bir bilgi olarak liste yapısı normalde her eleman için Veri ve bir sonraki elemanı gösteren Adres olarak tutulurken, Lisp dilinde Veri'nin tutulduğu bölgeye "Car", adresin tutulduğu bölgeye "Cdr" denir.

Lisp dilinde `cdr` fonksiyonu adres döndürür, `nthcdr` fonksiyonu ise parametre olarak verilen konum ve listeden, listenin o konumunun adresini verir. push

fonksiyonu ise oraya elemanı eklememizi sağlayan ana fonksiyondur. Örneğin bir listenin belli bir indeksine belli bir eleman ekleyeceğimiz bir fonksiyon yazacak olursak;

```
(Defun Ekle(liste konum eleman)
  (push eleman (cdr (nthcdr konum liste)))
  liste
)
```

Örneğin 1 2 3 4 elemanları olan *A* dizisinin 3. indeksine 8 rakamını eklemek istersek;

`(Ekle *A* 2 8)`

dediğimizde yeni listemiz 1 2 3 8 4 olur.

Görüleceği üzere 3. indekse eleman ekleyecekken konum olarak 2'yi verdik ki 3. indekse eleman ekleyelim. Bunun sebebi de nthcdr'den bize dönen adres konum olarak 2 verdiğimiz için 2. indeksteki cdr'nin adresi oluyor. Bu da bildiğimiz üzere aslında bir sonraki düğümün adresini tutan yer olduğundan 3. indeksi gösteriyor ve 3. indekse değeri eklemiş oluyoruz.

Verilen konumdaki elemanı listeden silmek istersek bunu `delete` fonksiyonu ile sağlayabiliriz.

```
(Defun Silme(liste konum)
  (delete (first (subseq liste konum)) liste)
  liste
)
```

first (subseq liste konum) bu kod verdiğimiz konumdan başlayarak örneğin 1 2 3 4 olan listede 3 elemanını silmek için konumu parametre olarak 2 (`(Silme *A* 2)`) verdiysek o konumdan başlayarak listeyi yeni bir şekle sokar yani 3 4 yapar ve bunun ilk elemanını döndürür ki bu da bizim silmek istediğimiz eleman olur.