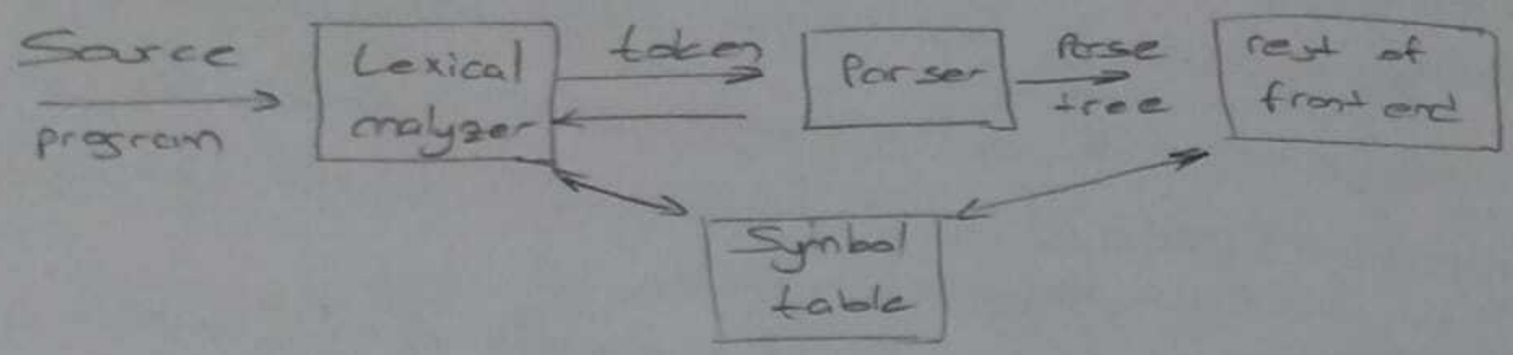
=> orta seviye kod (Semantik - analiz)



=> orta seviye kod -> Yüksek seviyeli dili düşük seviyeli dile çevirmek için makinenin anlayacağı dil için ara kod optimizasyon

-> Leftmost

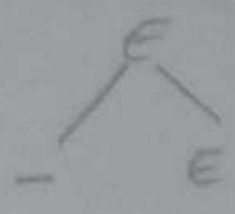
$E \rightarrow -E$
$E \rightarrow -(E)$
$E \rightarrow -(E+E)$
$E \rightarrow -(id+E)$
$E \rightarrow -(id+id)$

=> Rightmost

$E \rightarrow -E$
$E \rightarrow -(E)$
$E \rightarrow -(E+E)$
$E \rightarrow -(E+id)$
$E \rightarrow -(id+id)$

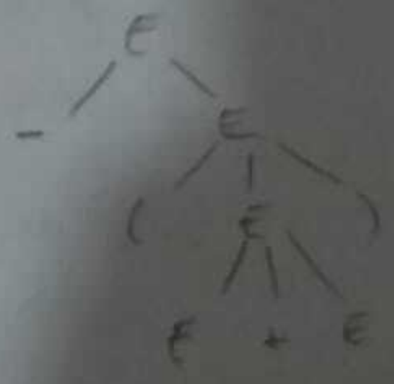
=> Parse Tree (Left most)

$E \rightarrow -E$

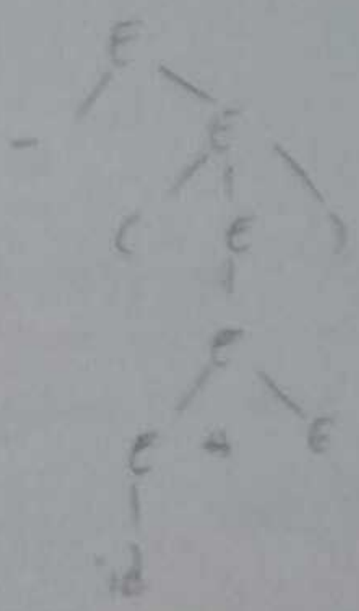


$\rightarrow -(E)$

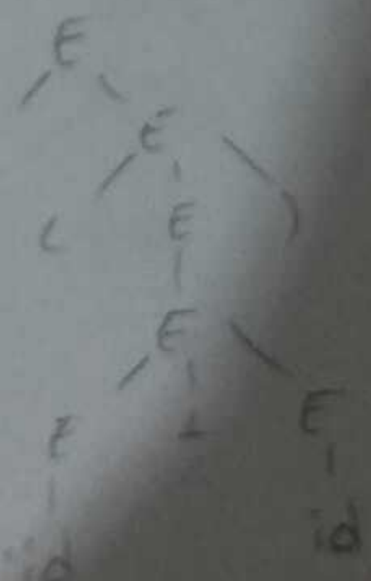
$\rightarrow -(E+E)$



$\rightarrow -(id+E)$

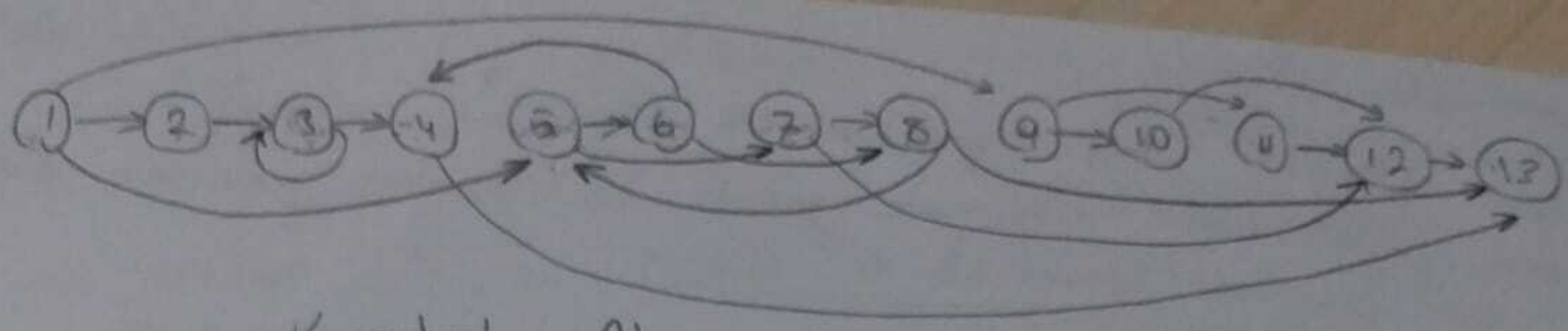


$\rightarrow -(id+id)$



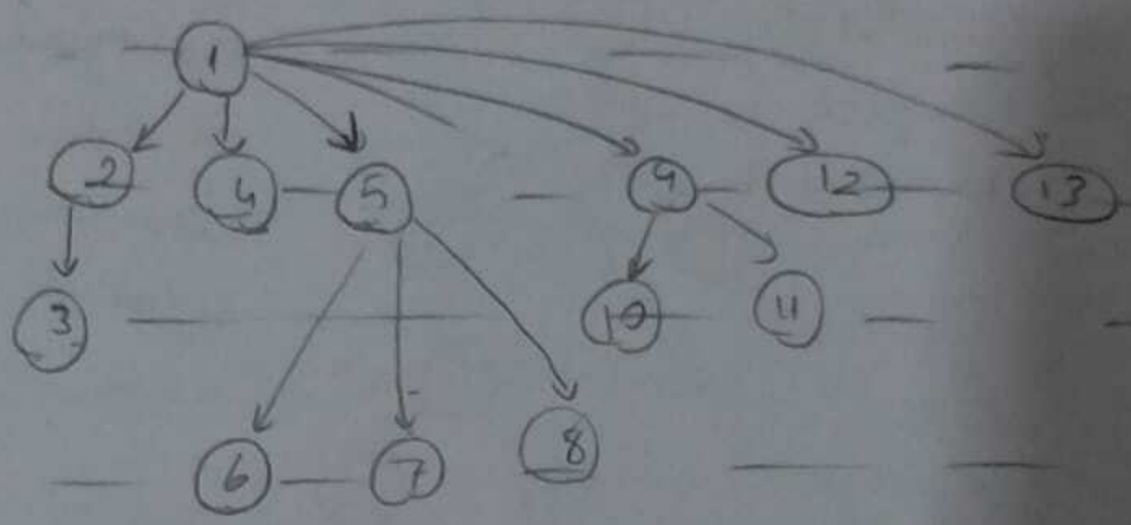
Left için Parse Tree oluşturma

Top-Down
recursive parsing
predictive parsing
program



Kontol Akış Grafı

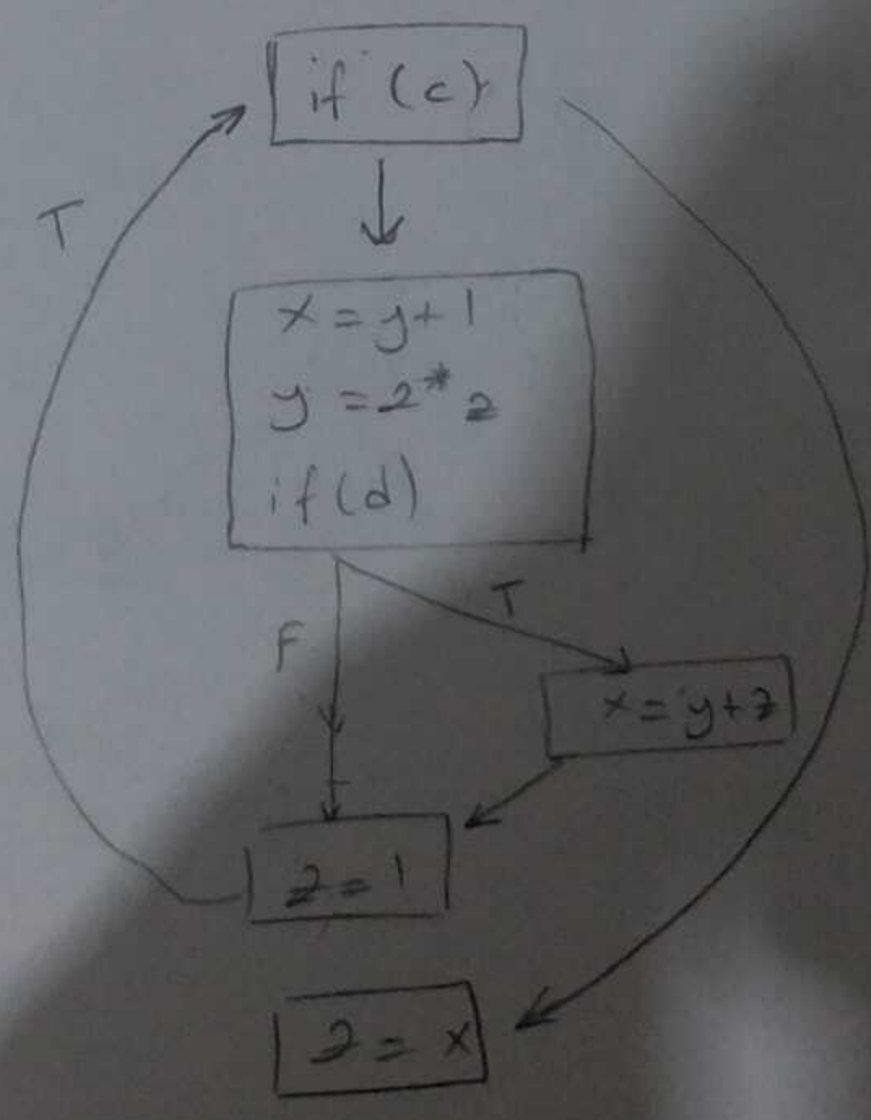
Yolların ortak düğümleri Dominat olur.



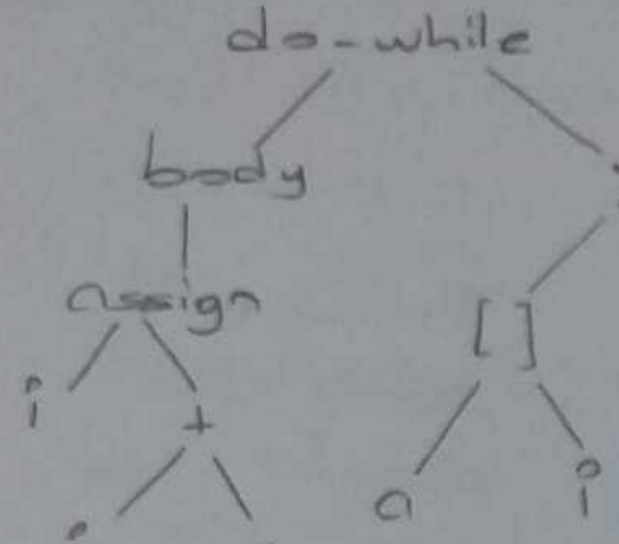
Dominatör Ağacı

=> while (c) { ←

```
x = y+1;
y = 2* z;
if(d) x = y+z;
z=1;
}
z=x;
```



F = false
T = True



(1) i = i+1
(2) $t_1 = a[i]$
(3) if $t_1 < v$ goto (1)

3 address kodu
7

→ Syntacks Tree

=> do i=i+1; while (a[i]<v); Yüksek seviyeli dil
7

=> x = y op z
=> x[y] = z
=> x = y[z]
=> if false x goto L
=> if True x goto L
=> goto L

3 address kodunun belli özellikleri
7

t_1
t_2
t_3
=> a[i] = 2 * a[j-k]

=> do i=i+1; while(a[i]<v);

$t_3 = j - k$
$t_2 = a[t_3]$
$t_1 = 2 * t_2$
$a[i] = t_1$

7

L: $t_1 = i+1$
$i = t_1$
$t_2 = i * 8$
$t_3 = a[t_2]$
if $t_3 < v$ goto L

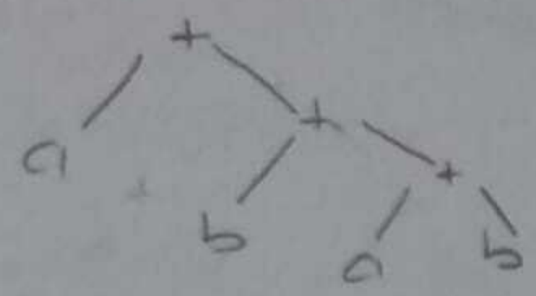
Tek sembol gös
7

100: $t_1 = i+1$
101: $i = t_1$
102: $t_2 = i * 8$
103: $t_3 = a[t_2]$
104: if $t_3 < v$ goto 100

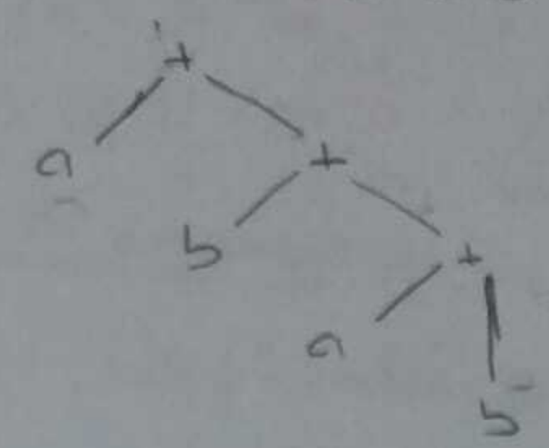
numaralı gösterim
7

=) Syntax Tree

=) a+b+(a+b)

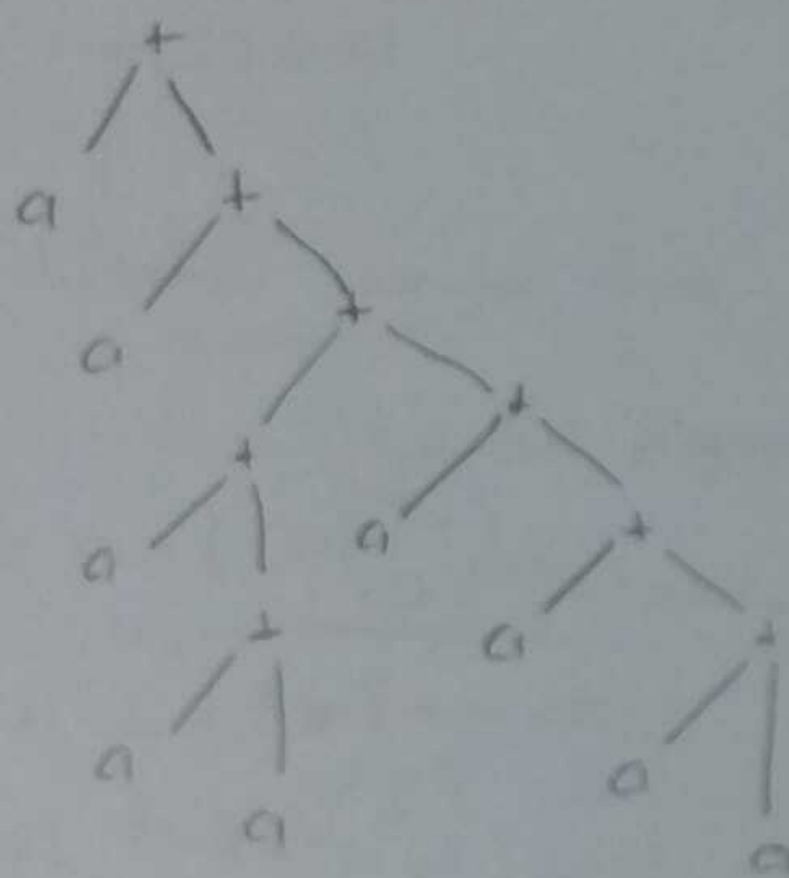


=) a+b+a+b



Yanlış olabilir araştır

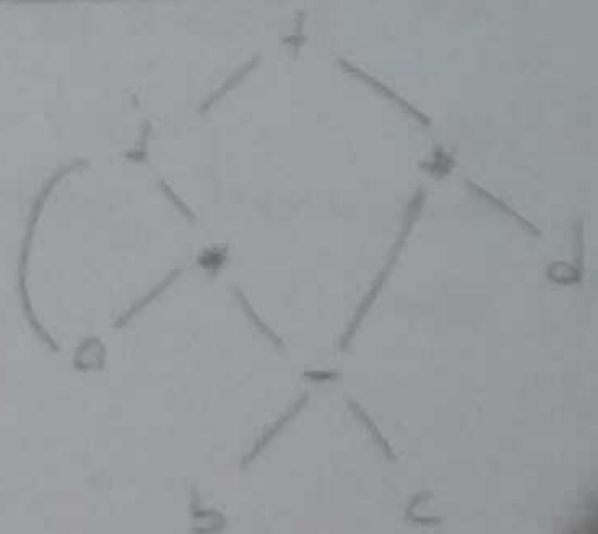
=) a+a+(a+a+a+(a+a+a+a))



Yanlış olabilir araştır.

Yüksek seviye dili parçalanır,

=) a+a*(b-c)+(b-c)*d



a) DAG

Syntax Tree

7

=) $t_1 = b - c$

$t_2 = a * t_1$

$t_3 = a + t_2$

$t_4 = t_1 * d$

$t_5 = t_3 + t_4$

3 address kodu olur

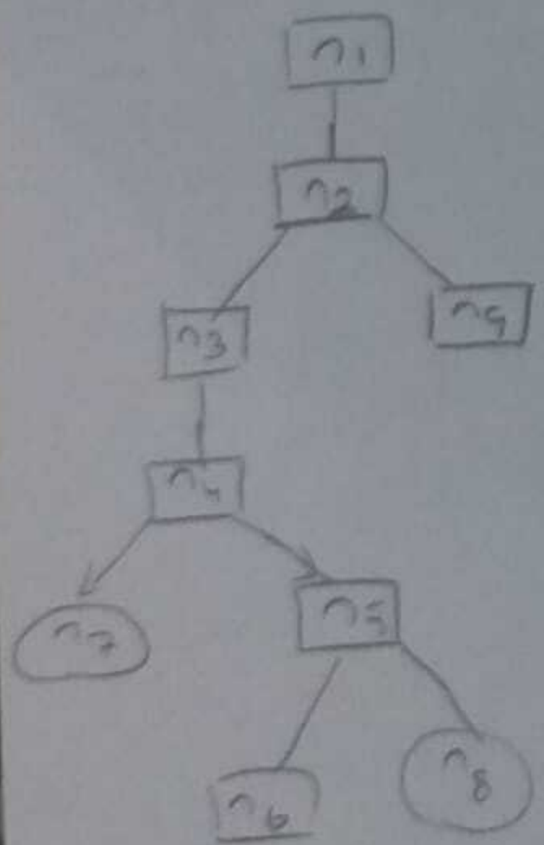
b) 3 adres kodu

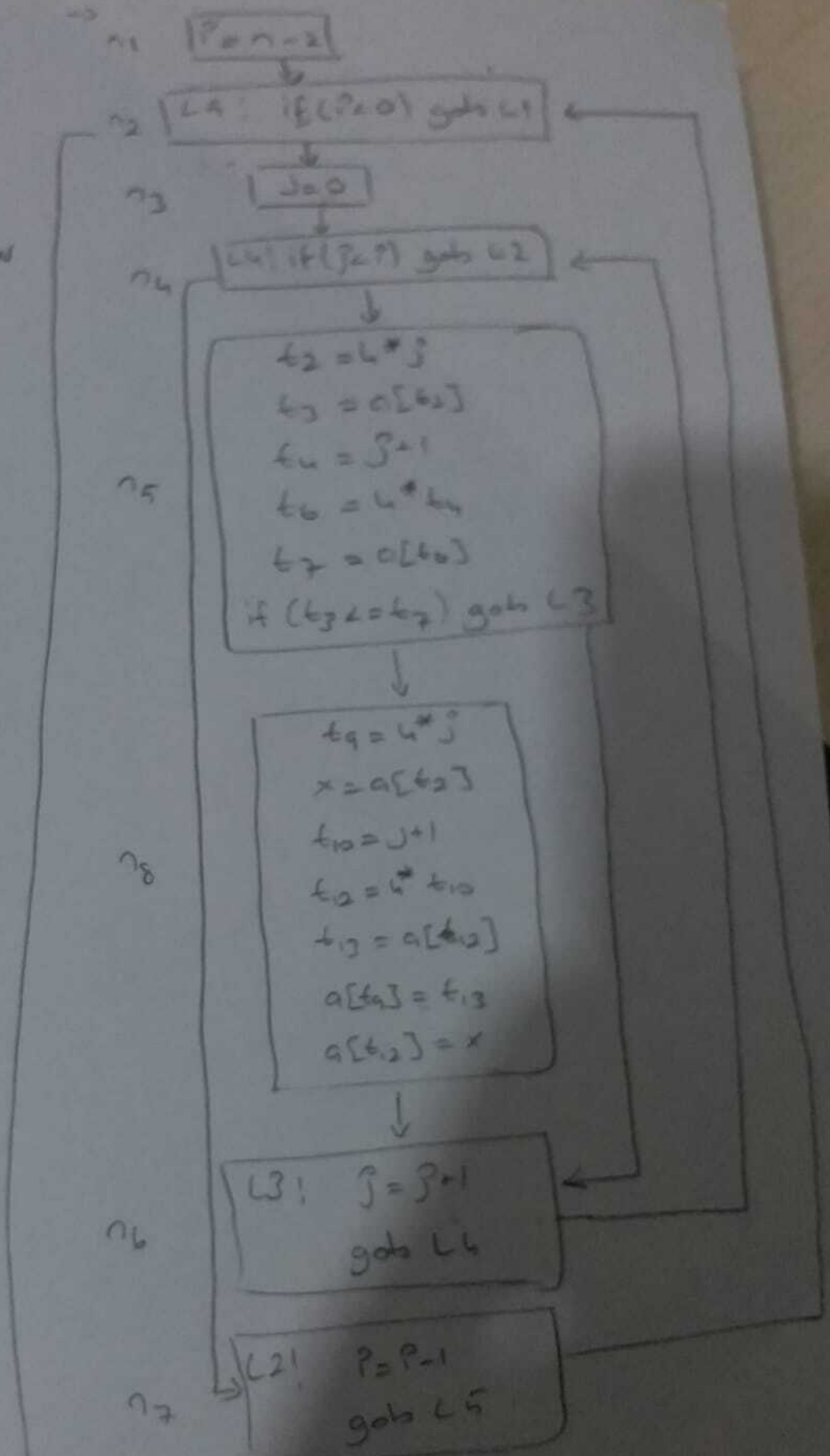
7

a) bubble sort kodu

```
FOR i:=n-1 DOWNTO 1 DO
FOR j:=1 TO i DO
if A[j] > A[j+1] THEN BEGIN
temp:=A[j];
A[j]:=A[j+1];
A[j+1]:=temp;
END
```

3 adresli kod
(Diğerlerine bak)
Verebilir
Vermeyebilir





Kalıp sorudu.
Diğerleri bakılacak

=> postfix Notation

=> ((A+B)*C)-((D+E)/F)) → (((3*2)/2)+(8/4)+2))

-> ((AB+*C)-((D+E)/F)) → ((32*/2)+((8/4)+2))

-> (AB+C*-((D+E)/F)) → (32*2/+((8/4)+2))

-> AB+C*(DE+/F)- → 32*2/(84/+2)+

-> AB+C*DE+F/- → 32*2/84/2++

Production	Semantik Rule	Program Fragment
E1 → E1 op E2	E.code = E1.code ‖ E2.code ‖ op	→ print op
E → (E1)	E.code = E1.code	
E → id	E code = id	→ print id

=> Three - Address Code

A = -B * (C + D)

↓

T1 = -B

T2 = C + D

T3 = T1 * T2

A = T3

sek. olur

?

(1) if A<B goto (4)

(2) T! = 0

(3) goto (5)

(4) T! = 1

(5)

↓

Yüksek seviyeli dili

(1) if A<B goto (4)

(2) T1 = 0

(3) goto 5

(4) T1 = 1

(5) T2 = T1 or C

Yüksek seviyeli dil

if (A<B) {

T = 1; }

else {

T = 0; }

end;

?

if (A<B) {

T = 1; }

else {

T = 0; }

end;

?

Yanlış olabilir araştır.

=> if (A<B || C<D) x = y + z

yeya

(1) if (A<B) goto 4

(2) if (C<D) goto 4

(3) goto (6)

(4) T = y + z

(5) x = T

Tek seferde yazılırsa 2 ye böl

?

for (i=0; i<n; i++)

x = x + 1;

3 adres kodu

(1) i = 0

(2) if (i<n) goto (5)

(3) goto (9)

(4) k = i + 1

(5) i = k

(7) x = T

(8) goto (2)

```
=> int i; int j;
   float[100] a; float v, float x;
   while(true){
   do
      i=i+1;
   while(a[i] < v);
   do
      j=j-1;
   while(a[j] > v);
   if(i>=j) break;
   x=a[i];
   a[i]=a[j];
   a[j]=x;
   }
```

4. Örnek soruya kod

```
=> n=f(a[i]);
```

```
1: i=i+1
2: t1=a[i]
3: if t1<v goto 1
4: j=j-1
5: t2=a[j]
6: if t2>v goto 4
7: iffalse i>=j goto 9
8: goto 14
9: x=a[i]
10: t3=a[j]
11: a[i]=t3
12: a[j]=x
13: goto 1
14:
```

3 address kodu

alt program

```
1: t1=i*4
2: t2=a[t1]
3: param t2
4: t3=call f,1
5: n=t3
```

Temel blok

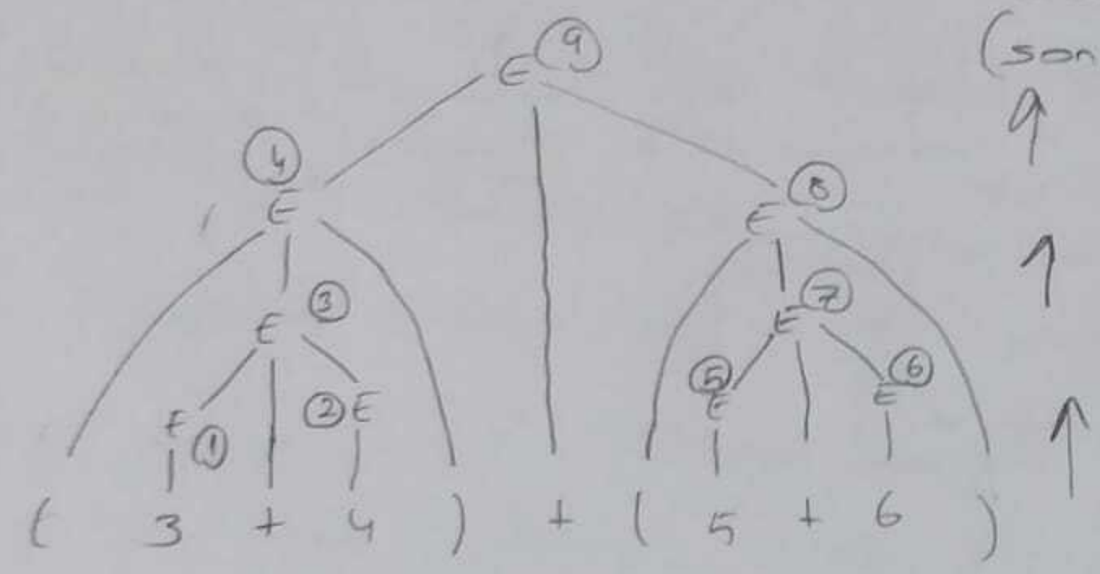
1) Orta koddaki ilk satır lider-dir.
2) Şartlı veya şartsız bir goto hedefteki lider-dir.
3) Şartlı goto'dan sonraki gelen satır lider.

```
→ (1) i=1
→ (2) j=1
→ (3) t1=10*i
  (4) t2=t1+j
  (5) t3=8*t2
  (6) t4=t3-88
  (7) a[t4]=0.0
  (8) j=j+1
  (9) if j<=10 goto (3)
→ (10) i=i+1
  (11) if i<=10 goto (2)
→ (12) i=1
→ (13) t5=i-1
  (14) t6=88*t5
  (15) a[t6]=1.0
  (16) i=i+1
  (17) if i<=10 goto (13)
```

```
for i from 1 to 10 do
  for j from 1 to 10 do
    a[i,j]=0.0;
for i from 1 to 10 do
  a[i,i]=1.0;
```

⇒ LL parsers (LL - aşağı)

aşağıdan - yukarıya (bottom-up)
(sondan - başa)



Bu karmaşık işlemler için DFA çizilir böyle bulunur
Doğrulamak için

-> LR(0)

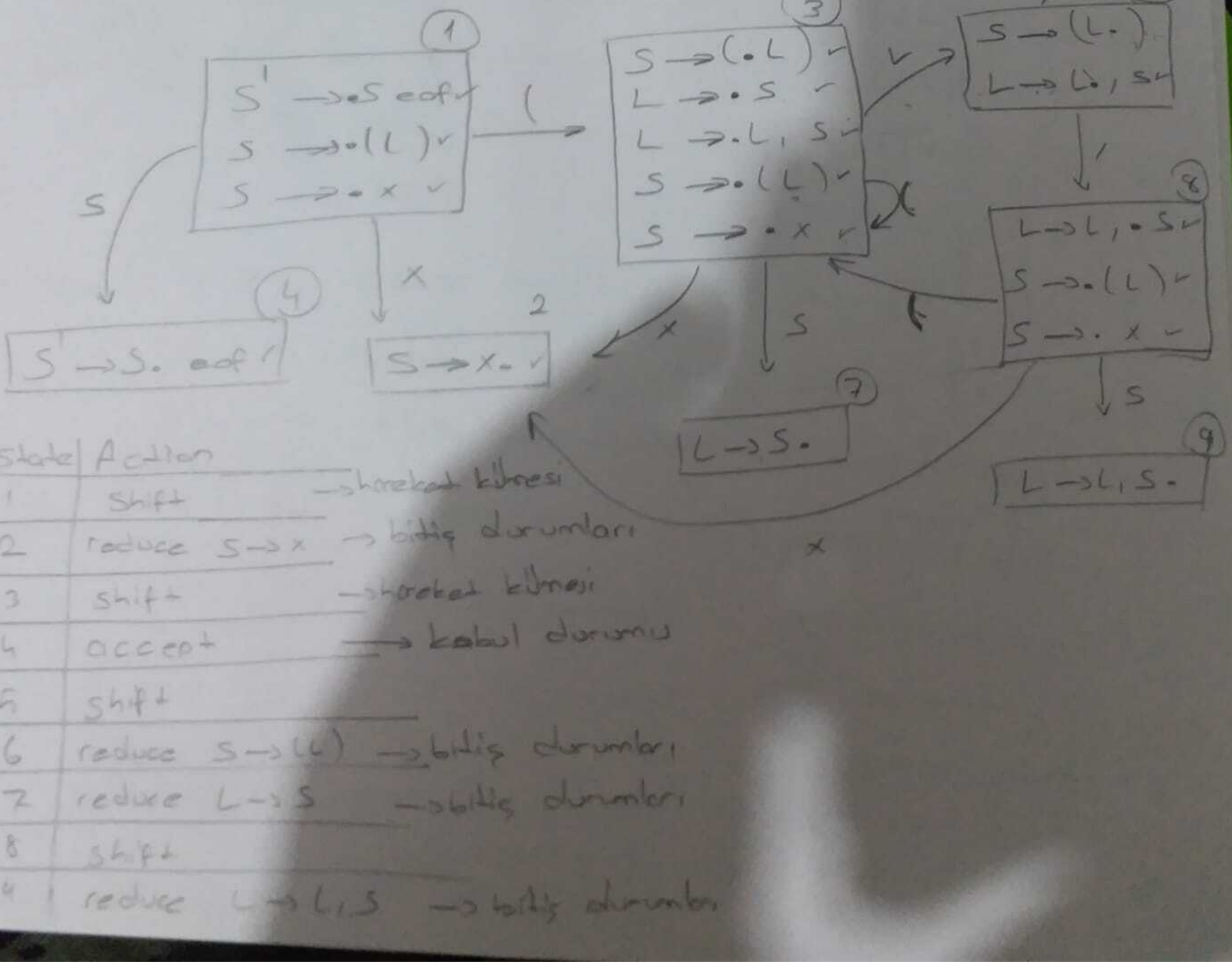
Gramer düzen

S → (L)
S → x
L → S
L → L, S

DFA için =>

S' → S eof
S → (L)
S → x
L → S
L → L, S

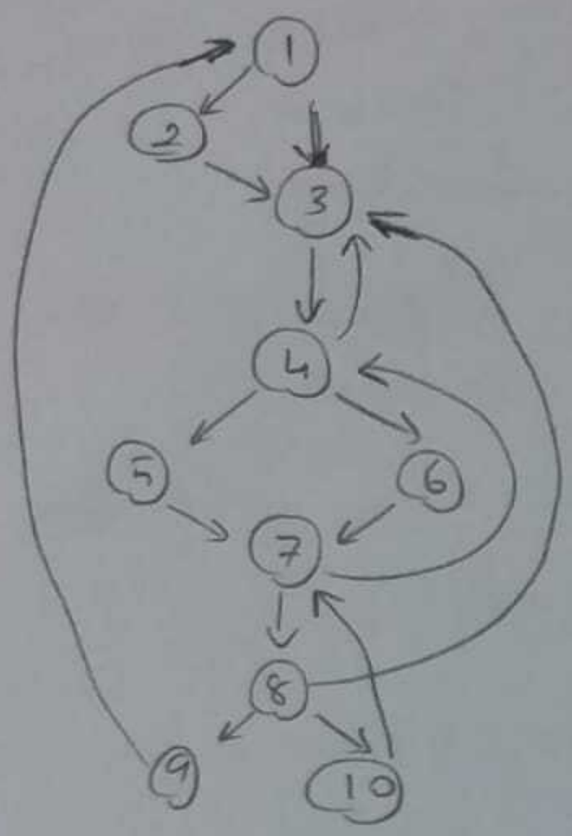
-> nokta koy noktadan sonra gelen karakter kümeyi oluşturur veya başka bir durumu ifade eder ise onuda dahil et |S → (L).



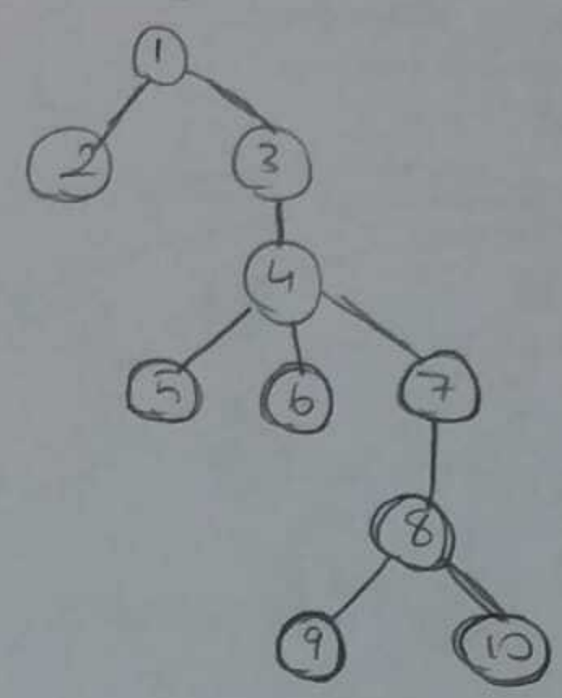
State	Action	
1	shift	→ hareket kümesi
2	reduce S → x	→ bitiş durumları
3	shift	→ hareket kümesi
4	accept	→ kabul durumu
5	shift	
6	reduce S → (L)	→ bitiş durumları
7	reduce L → S	→ bitiş durumları
8	shift	
9	reduce L → L, S	→ bitiş durumları

⇒ Dominatör Ağacı

⇒ Eğer Kontrol akış grafında n düğümüne gitmek için başlangıç düğümünden geçmek şartı ile d'den geçmek zorun lüğü var ise d dominat n denir.

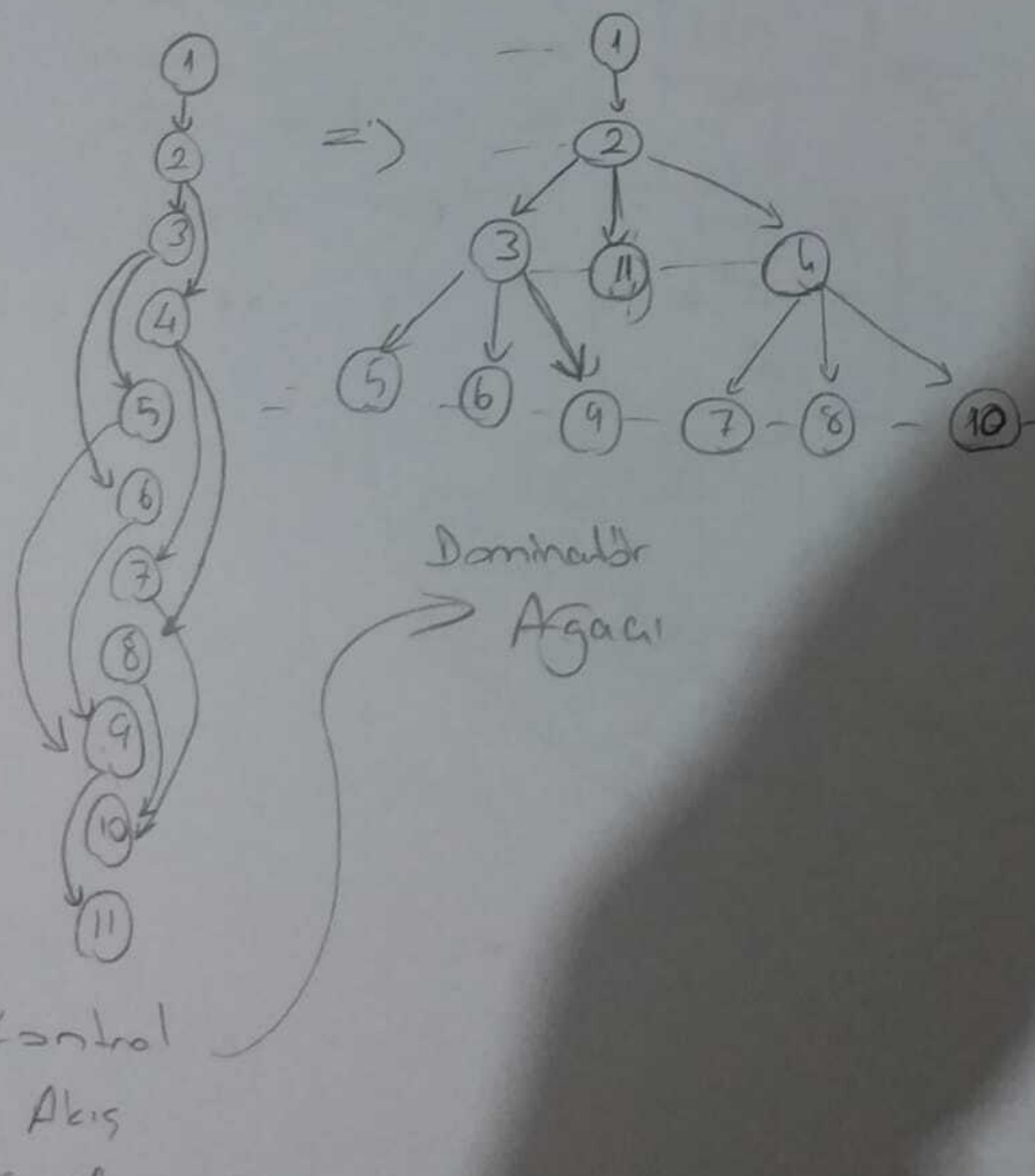


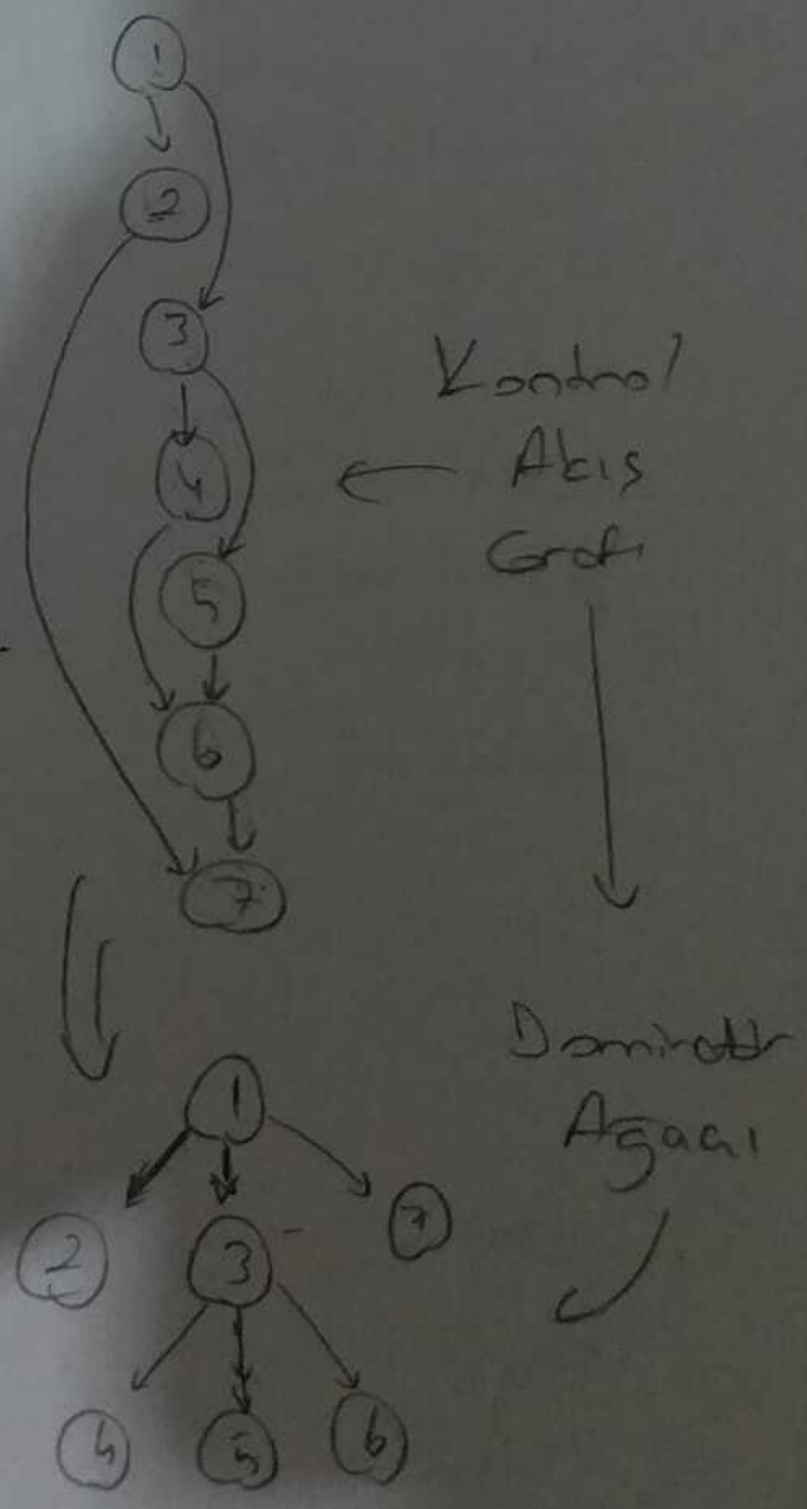
Kontrol Akış Graf.



Dominatör Ağacı

=> 7 → 5 ve 6 dan geçiyor ikisinde aynı seviye olunca ikisinde 4'den geçme zorunlu ise 4'ün altına yazılır.





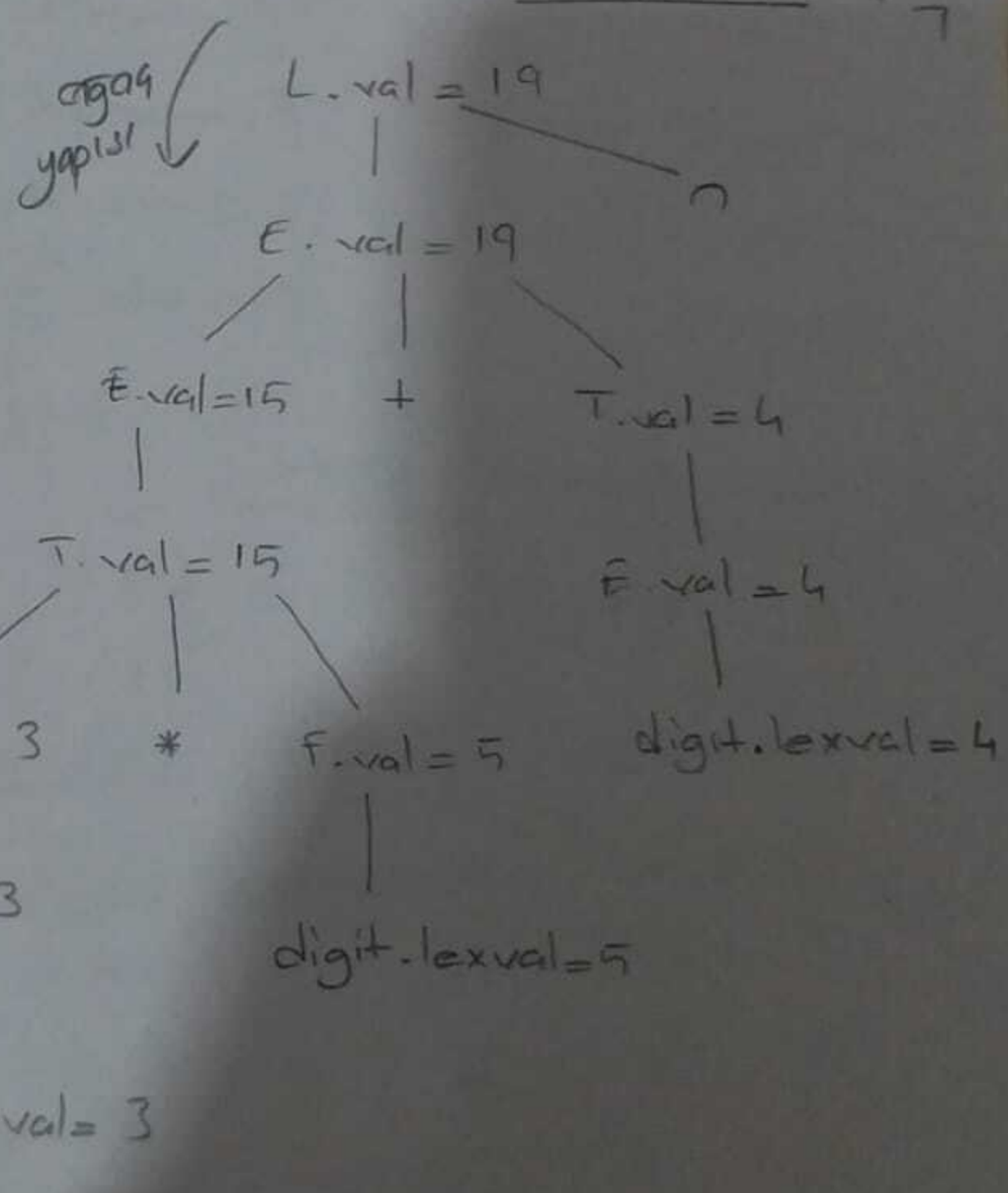
=> Bir hesap makinesinin + ve * işlemlerini yapıp ekranda sonuna n harfini girip enter denilir ise hesaplama yapması isteniyor.

Production	Semantic Rules
1) L → E n	L.val = E.val
2) E → E1 + T	E.val = E1.val + T.val
3) E → T	E.val = T.val
4) T → T1 * F	T.val = T1.val × F.val
5) T → F	T.val = F.val
6) F → (E)	F.val = E.val
7) F → digit	F.val = digit.lexval

Not: Derleyici mutlaka grameri tanıması gerekir yoksa işlem yapmaz.

Gramer → (digit yapılan (F) yani E için ara işlemleri yapmalı) → derleyici yapısı

Parsetree → 3*5+4n işlemi → derleyici yapısı



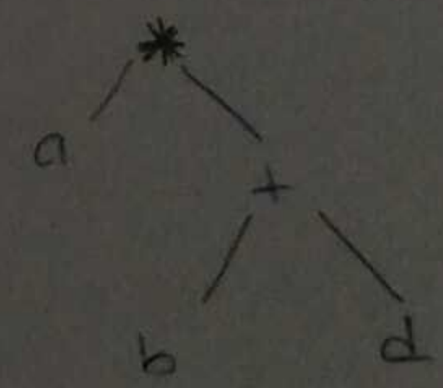
=> Orta seviye Kod Üretimi

-> Syntax trees

-> postfix notation

-> three - addres code

a*(b+d)



=> Syntax Tree

Production	Semantik Rule
E → E1 op E2	E.val = NODE (op, E1.val, E2.val)
E → (E1)	E.val = E1.val
E → -E1	E.val = UNARY (-, E1.val)
E → id	E.val = LEAF (id)

=> → a=b=c=1
x[*]=0
} B_1

→ L1: if() goto L3 } B_2

→ a=a+1
b=b+1
goto L4
} B_3

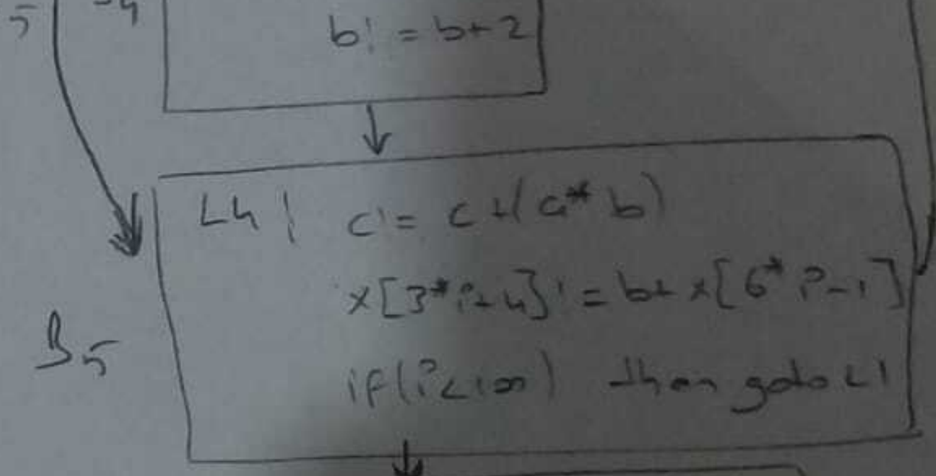
→ L3: a=a+2
b=b+2
} B_4

→ L4: c=c+(a*b)
x[3*i+4]=b+x[6*i-1]
if (i<100) then goto L1
} B_5

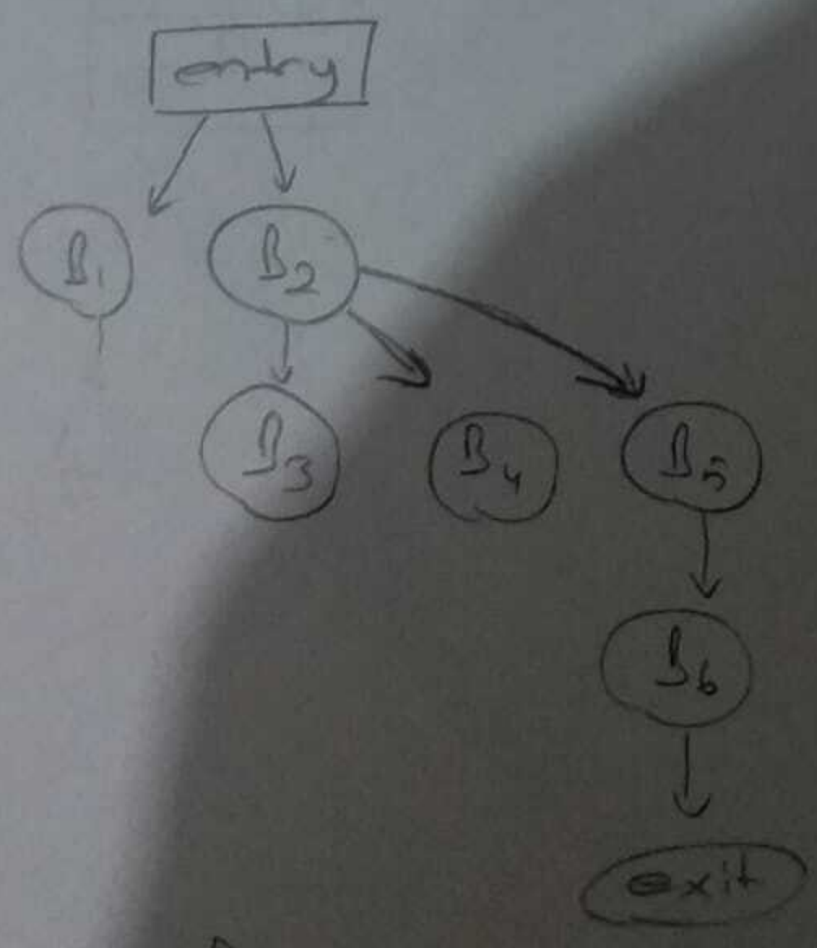
→ print(c, x[100]) } B_6

Gikmis Soru

-> lider bul -> sardi gotu lidler.
ve sonraki satirlar lider.



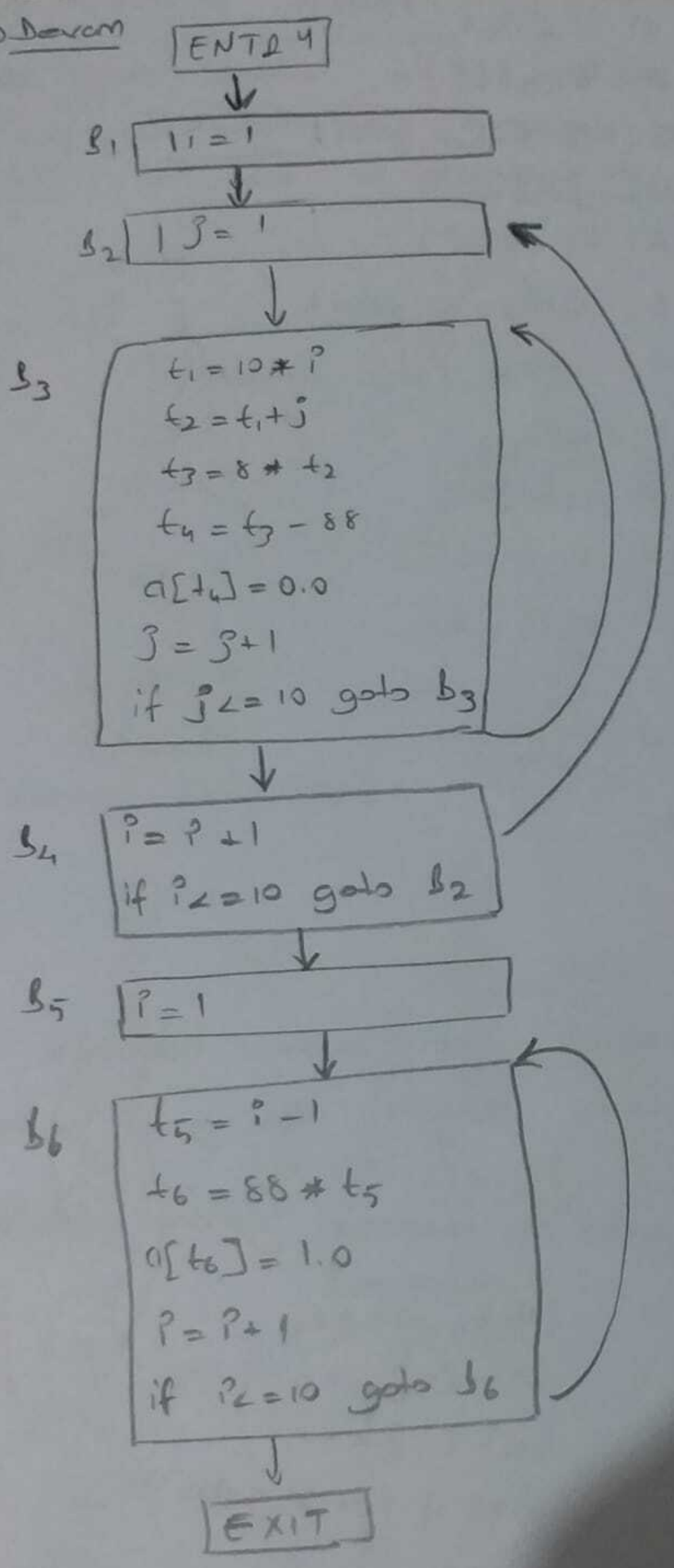
Kontrol Akis Grafi



Dominator Agaci

Cevap yanlis olabilir Arastirin

=> Devam



Kontrol Akış Grafları

=> Derleyici Tasarımında ara seviye kod üretimi yapıldıktan sonra bu kodun analizini yapmak için blok seklinde (tabloar) parcalara ayrılıp akışa dökülmüş halidir.

=> liderleri belirle

=> liderler arasındaki satırlar temel blokları oluşturur.

=> Bu blokların akışınada Kontrol Akış Grafı denir.

liderlerin bloklara ayrılması

blok şeması

Kontrol Akış Grafiği

(Sınavda çıkabilir.)

=> Syntax Directed Translation

Production	Semantic Rule	Program Frame
L -> E return	print (E.val)	print (val[top-1])
E -> E' + T	E.val = E'.val + T.val	val[ntop] = val[top-2] + val[top]
E -> T	E.val = T.val	
T -> T' * F	T.val = T'.val * F.val	val[ntop] = val[top-2] * val[top]
T -> F	T.val = F.val	
F -> (E)	F.val = E.val	val[ntop] = val[top-1]
F -> digit	F.val = digit.lexval	val[top] = digit.lexval

=>

Production	Semantic Action
S -> id := E	S.code = E.code || gen(id.place = E.place)
E -> E1 + E2	E.place = newtemp(); E.code = E1.code || E2.code || gen(E.place = E1.place + E2.place)
E -> E1 * E2	E.place = newtemp(); E.code = E1.code || E2.code || gen(E.place = E1.place * E2.place)
E -> - E1	E.place = newtemp(); E.code = E1.code || gen(E.place = - E1.place)
E -> (E1)	E.place = E1.place E.code = E1.code
E -> id	E.place = id.place E.code = null

LL(k)

LL(k)
- Left-to right-to (sola dayalı sağa dayalı) — Soldan sağa oku
- Rightmost derivation (sağdan türetme)
- k - symbol (k kadar ileri bak)

bir

=> E → int (Contex free Gramer) → Derleyicinin 2. aşaması → kaynak kodu tanımlamanın sağlar. (CFG) (Sytanx analizi)

E → (E)

E → E+E

Yığıt → (ε kalana kadar devam)

Giriş (Input)	Yığıt	Hareket (Action)
(3+4)+(5+6)	(	→ shift → E → (E)
3+4)+(5+6)	(3	→ Shift → E → (E)
+4)+(5+6)	(3	→ Reduce → E → int → 3
+4)+(5+6)	(E	→ Shift → E → E+E
4)+(5+6)	(E+	→ Shift → E → int
)+(5+6)	(E+4	→ reduce → E → int → 4
)+(5+6)	(E+E	→ reduce → E → E+E
)+(5+6)	(E	→ shift → E → (E)
+(5+6)	E	→ reduce → E → (E) → E
+(5+6)	E	→ shift → E+E → devam
(5+6)	E+	→ Shift
5+6)	E+(	→ Shift
+6)	E+(5	→ reduce E → int → 5
+6)	E+(E	→ shift
6)	E+(E+	→ Shift
)	E+(E+6	→ reduce E → int → 6