Kaynak: Compilers Principles Techniques and Tools

Çarşamba 15:00

= DERLEYİCİ TASARIMI =

26.02.2014
Çarşamba

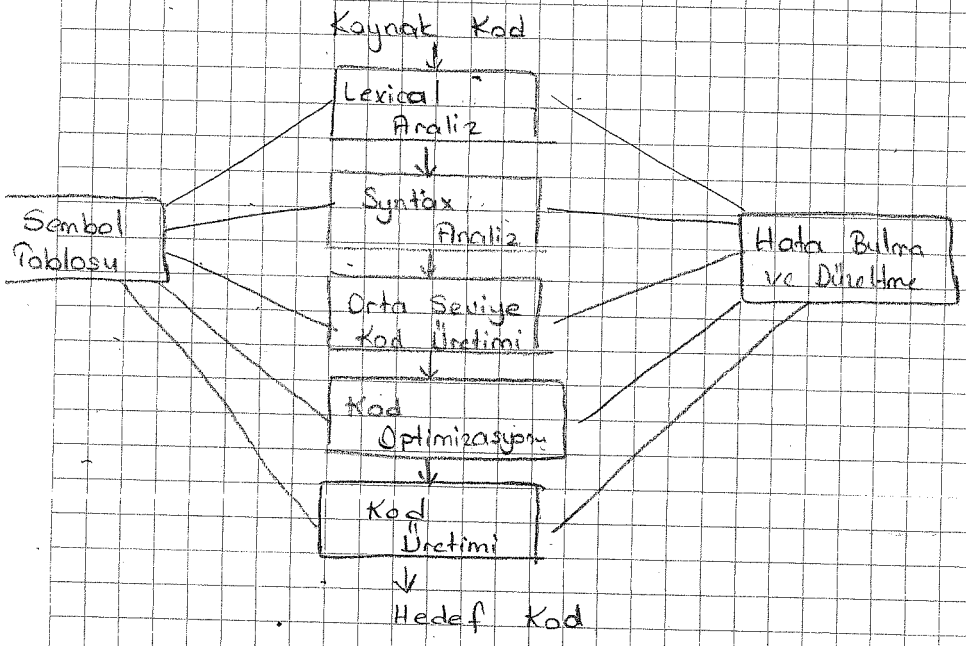
Derleyici (Compiler) = Yüksek seviyeli dili makina koduna çevirirler

Dönüştürücüler (Translaters) = Yüksek seviyeli bir dili başka bir yüksek seviyeli dile çevirirler

İlk derleyicilerden ; Fortran, ...

Fortran derleyicisinin yapılması 18 yıl almış.

Derleyicinin Yapısı



Lexical Analiz Aşamasında, kaynak kodu en küçük birimlerine ayırır.

ÖRN

```
if(a<b)
  a=a+b
```

=> if (a < b = + — Dilin en küçük parçalarına ayırdı

Syntax Analiz Aşamasında, söz dizimsel analiz yapar. Lexical analiz sonucu elde edilen token ların dilin kurallarına göre doğru sıralanıp sıralanmadığını kontrol eder.

Orta Seviye Kod Üretimi Aşamasında, derleyiciyi kullandığımız yüksek seviye dil ile makina dili arasında orta seviye bir dil üretir. if ve go to'lardan oluşan bir koda dönüştürür.

Kod Optimizasyonu Aşamasında, gereksiz kodlar, ilişkiler, yapılabilecek kısaltmalar vb. çeşitli algoritmalarla gerçekleştirilir.

Kod Üretimi Aşamasında, elde edilen kod makina kodunu(diline) dönüştürür.

Sembol tablosu, yazdığımız kaynak kod içerisinde birçok değişken var, kompleks bir yapı oluşturuyor. Sembol tablosu da bu kamasık yapıdaki sembollerin, değişkenlerin nerede, nasıl çalıştığını gözlemek, durumunu gözden geçirmek için kullanılır.

(Orta seviye kod üretimi)

ÖRN.

```
while (A>B & A<=2*B-5)
        A=A+B;
```

```
L1 : if A>B go to L2
     go to L3
L2 : T1 = 2*B
     T2 = T1-5
     if A<=T2 go to L4
     go to L3
L4 : A = A+B
     go to L1
L3 : ...
```

ÖRN if A>B go to L2 } if A<=B go to L3
go to L3

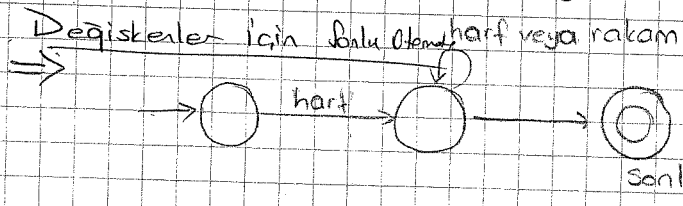
ÖRN A = B+C } LOAD B
ADD C
STORE A

Lexical Analiz

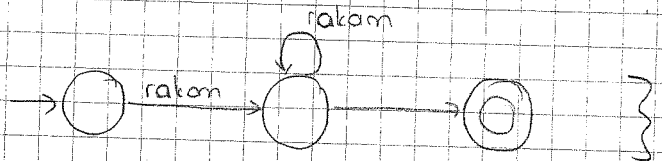
Bu aşamada token'ların bulunmasını sağlayan otomatadır.
Biz bir lexical analiz oluşturmak istersek, karakter karakter okuruz. Pointer kullanarak yapabiliriz.

ÖRN deneme
↑ - - - ↑
Birinci pointer başlangıcı tutar
→ İkinci pointer karakterleri okur ve karakterde kelimelerin oluşturduğu dizilimin istenilen şekilde olduğunu görünce durur :)
İki pointer kullanıldı

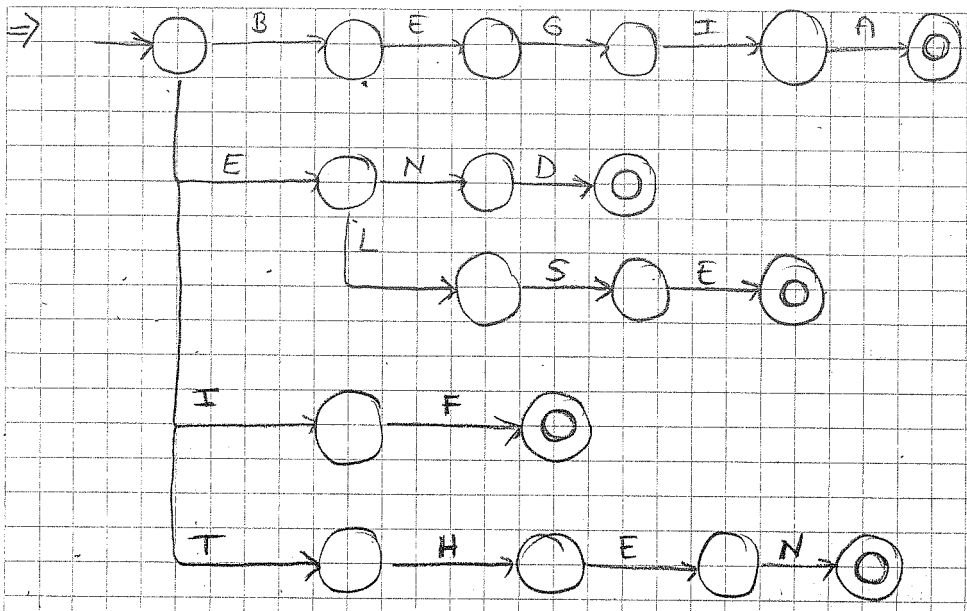
Ödev: Lexical analiz aşamasını gerçekleştiren bir kod parçası yaz.



Lexical analiz yukarıdaki gibi birçok otomata içerir.



} rakam için sonlu otomata

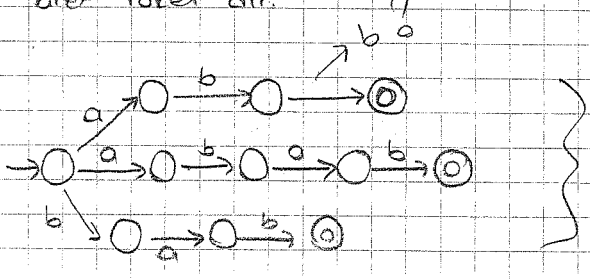


⇒ ilk olarak derleyicinin alfabesi (karakterleri), daha sonra dil bilgisine bakılır. Derleyici tasarımında alfabe olmazsa olmazdır. Derleyici tasarlamaya ordan başlanır.

* String, alfabenin karakterlerinden oluşan yapıdır
* Dil, stringlerden oluşan bir kümedir
* Her dilde mutlaka alfabe vardır, her dilin bir gramer vardır
* Her bir string birer token'dır.

ÖRN

abb
abab
bab

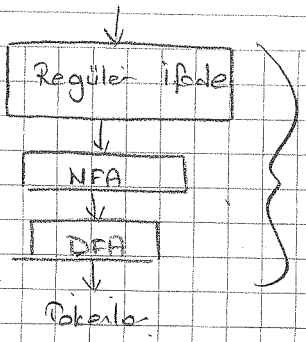


Verilen 3 token'ı tanıyan DFA

3e+10 ⇒ 3.10^10

3E10

05.03.2014



⇒

letter = A|B|...|Z ⇒ a+b+c+...+z

digit = 0|1|...|9

identifier = letter (letter | digit)*

integer = digit+ (digitlerden en az bir tane gelecek, daha sonra daha fazla gelebilir)

sign = +|-|E

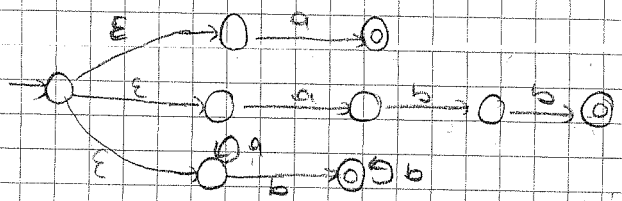
Signed integer = sign integer

ÖRN

a

abb

a*b+

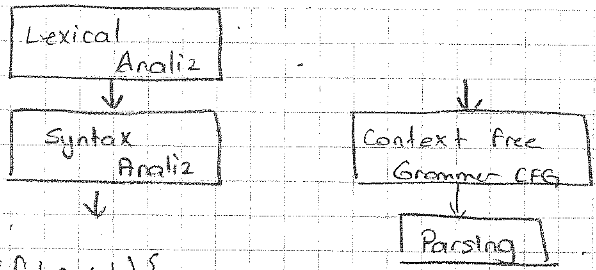


Token'larına ayırmak için algoritmalar kullanırız.

Derleyicinin boyutu büyüdükçe sistematik bir şekilde yapılması gerekiyor.

Bunun için de algoritmalar ve DFA'lar kullanırız.

Parse → syntax analiz kısmında gerçekleştirilir.




```
if (a < b) {
    a = a + b
    c = a + c
}
```

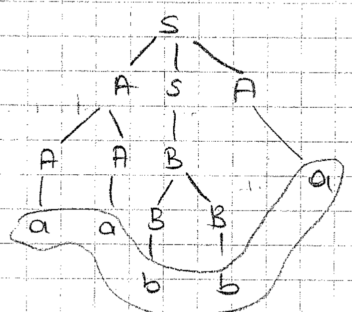
(söz dizim değişkenleri, uç simge, kurallar, start sembolü)

⇒ CFG : $< V_N, V_T, P, S >$

S ⇒ A | B | ASA
A ⇒ AA | a
B ⇒ BB | b

* Syntax analizi, gramerler yardımı ile gerçekleştirilir (yukarıdaki CFG'deki gramer)

S
A S A
A B B a
aabba

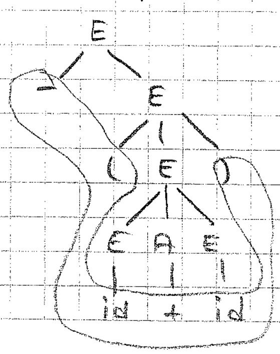


ÖRN

E ⇒ EAE | (E) | -E | id
A → + | - | * | /

} Buradan E ⇒ -(id+id) elde edebilir miyiz?

E ⇒ -E ⇒ -(E) ⇒ -(EAE) +
⇒ -(E+E) ⇒ -(id+id)



LR Parsing (Left to Right)

3 tipi vardır

* Syntax analiz yapmak için kullanılan yöntemlerden biridir.

Simple LR Parsing
Canonical LR Parsing

Canonical LR Parsing, geniş gramerlerle çalışır. Gerçekleştirilmesi zor.
Simple " = küçük " " " kolay
En sondaki , ikisinin arası

ÖRN
$E' \Rightarrow E$
$E \Rightarrow E+T \mid T$
$T \Rightarrow T*F$
$F \Rightarrow (E) \mid id$

Böyle bir gramerde parsing işlemini nasıl yaparız?
⇒ Bu tarz sorular çözerken aşağıdaki iki fonksiyonu kullanırız

1) CLOSURE(I) = Bir kez uygulanıyor. Başlangıç kümesini oluşturur.
2) GO TO (I, X) = Ondan sonraki tüm kümeleri oluşturmak için kullanılır.

I_0:
$E' \to .E$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$

} Closure fonksiyonu uygulandıktan sonra

2.

I_1: $E' \to E.$, $E \to E.+T$ } Goto (I_0, E)

E'den önce dön noktalar bir sonraki gidecek

I_2: $E \to T.$, $T \to T.*F$ } Goto (I_0, T)

I_3: $T \to F.$ } Goto (I_0, F)

I_4:
$F \to (.E)$
$E \to .E+T$
$E \to .T$
$T \to .T*F$
$T \to .F$
$F \to .(E)$
$F \to .id$
} Goto$(I_0, ($

yeni oluştu

Not: Önceki durumlara bakıyoruz. Eğer I_0 içindeki kurallar içeriyorsa aynı kümeye ekliyoruz.

I_4'ü oluştururken I_1, I_2, I_3'le bakıyoruz. . 'dan sonra gelen (varsa uyguluyoruz

NOT Bütün kümelerde bütün noktalar sona kadar gitmeli. Dizilimi bilebilmemiz için yeni küme oluşuyorsa olmalı (en sondaki otomata)

Buradaki I_2'yi veriyor

I_5: $F \to id.$ } Goto (I_0, id)

I_{12}: $T \to T.*F$, $E \to T.$ } Goto (I_7, I)

I_6: $E \to E+.T$, $T \to .T*F$, $T \to .F$, $F \to .(E)$, $F \to .id$ } Goto $(I_1, +)$

I_7: $T \to T*.F$, $F \to .(E)$, $F \to .id$ } Goto $(I_2, *)$

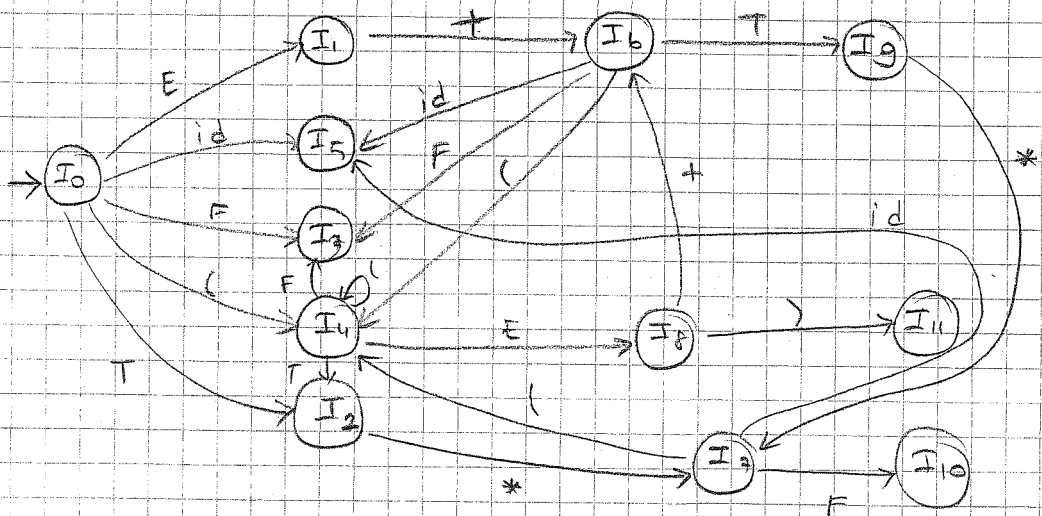
I_8: $F \to (E.)$, $E \to E.+T$ } Goto (I_4, E)

I_9: $E \to E+T.$, $T \to T.*F$ } Goto (I_6, T)

I_{10}: $T \to T*F.$ } Goto (I_7, F)

I_{11}: $F \to (E).$ } Goto $(I_8,)$

Bu türettiğimiz kümeleri ele alarak karşılık gelen otomatayı oluşturuyoruz.



Syntax ağacı: Sadece operatör ve karakterlerden oluşur

ÖRNEK $E \to E+T \mid T$

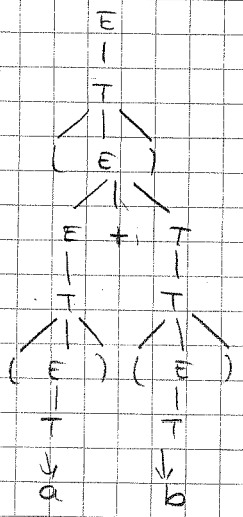
$T \to (E) \mid id$

$w = ((a)+(b))$ → tanır mı?

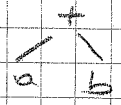
Parse ağacı ve syntax ağacı?

$E \Rightarrow T \Rightarrow (E) \Rightarrow (E+T) \Rightarrow (T+T) \Rightarrow ((T)+(T)) \Rightarrow ((a)+(b))$

Parse ağacı



Syntax ağacı



ÖRNEK

$S \to iSoT \mid o$

$T \to eSE \mid E$

$E \to b$

$w = iooeioob$ → tanır mı? (Tanır) ... Parse ağacını çiz

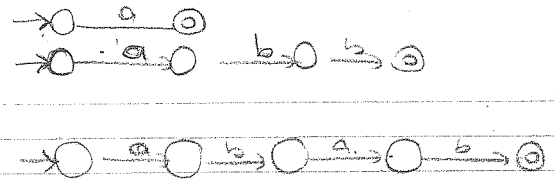
S
iSoT
iSoeSE
iSoeSb
iooeSb
iooeiSoTb
iooeioobTb → ?

20 puan

ÖRNEK

Token'lar tanıyan NFA ?

- a
- abb
- $a^* b^+$
- abab



40 puan

ÖRNEK $Z \rightarrow E\$$
$E \rightarrow T \mid E+T$
$T \rightarrow i \mid (E)$

→ LR Parsing diyagramını çiziniz. } 10 durum çıktı.

→ i+i 'yi tanır mı?

I_0 :

ÖRNEK

$S \Rightarrow SS1 \mid 0 \mid \varepsilon$ Gramere karşılık gelen regüler ifade=?

$\varepsilon, 0, 00, 000, 0000 \ldots \Rightarrow 0^*$

12.03.14

TEKRAR

Yüksek seviyeli dilde yazdığımız kod → Kaynak kod

Dili tokenlere ayırma → Lexical Analiz

Token'ların doğru sıralanıp sıralanmadığı → Syntax Analiz

* LR Parsing token'ların doğru sıralanıp sıralanmadığını kontrol etme yöntemlerinden biridir.

ÖRN

$S \Rightarrow AlaA$
$A \Rightarrow b|AA$

Buna karşılık gelen LR parsing? } CLOSURE GO TO komutlarını uygulayorduk.

⇒

NOT :
→ Küçük a'dan sonra mutlaka A gelmeli, gelmezse yanlış yapmışızdır.
Dizilim, sırlama önemli

Soru/ S ⇒ A | aA
A ⇒ b | AA

I_0: S ⇒ .A
S ⇒ .aA
A ⇒ .b
A ⇒ .AA

GoTo(I_0, A)
I_1: S ⇒ A.
A ⇒ A.A
A ⇒ .b
A ⇒ .AA

GoTo(I_0, a)
I_2: S ⇒ a.A
A ⇒ .b
A ⇒ .AA

GoTo(I_0, b)
I_3: A ⇒ b.

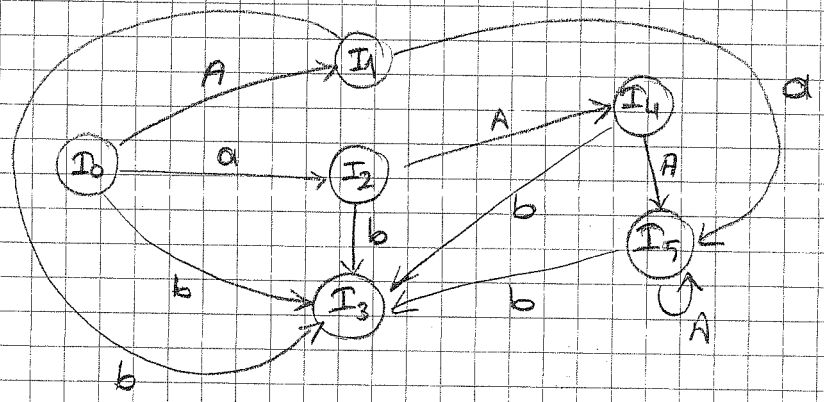
GoTo(I_2, A)
I_4: S ⇒ aA.
A ⇒ A.A
A ⇒ .b
A ⇒ .AA

GoTo(I_2, b) = I_3

GoTo(I_1, A)
I_5: A ⇒ AA.
A ⇒ A.A
A ⇒ .b
A ⇒ .AA

GoTo(I_4, A) = I_5

GoTo(I_5, A) = I_5 ...



* I_2 'den I_3'e giderken b var. Bu A ⇒ b olabilme kuralından geliyor.

S
A
AA
AAA
AAb

} Dizilimleri bu şekilde arttırabiliriz.

ÖRN for(i=0; i<5; i++)

Token 'lara ⇒ (Lexical analiz)

for
(
i
=
0
;
<
5
++

NOT Derleyicinin kurallarında + ile ++ ayrı token 'lardır.

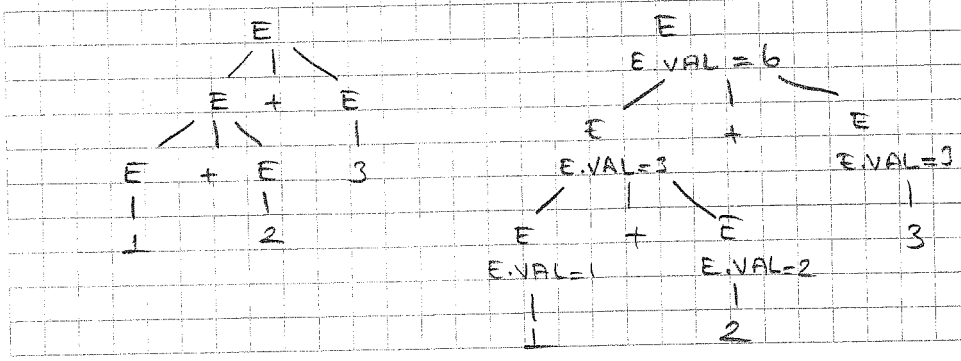
Syntax analiz ⇒ ilk ; 'de hata verir.
(DFA diagramından kontrol eder)

⇒ Semantik hata, anlamsal hatadır. Lexical, syntax açısından hatası olmayıp semantik hata olabilir. Daha sonrasında bir de zamansal (Run-Time) hata vardır.

Semantik Hareketler

Production	Semantik Hareket
$E \rightarrow E^{(1)} + E^{(2)}$	$E.VAL = E^{(1)}.VAL + E^{(2)}.VAL$
$E \rightarrow digit$	$E.VAL = digit$

ÖRN 1 + 2 + 3 girişi için



digit = 0, 1, 2, ... vb. karakterlerdir.

* CFG'yi kendi kurallarımızı yansıtacak şekilde iyi oluşturmalıyız.

ÖRN + ve * operatörlerini kullanan ve giriş stringleri $ ile sonlanan bir hesap makinası düşünelim. Sayısal değeri çıkışı versin.

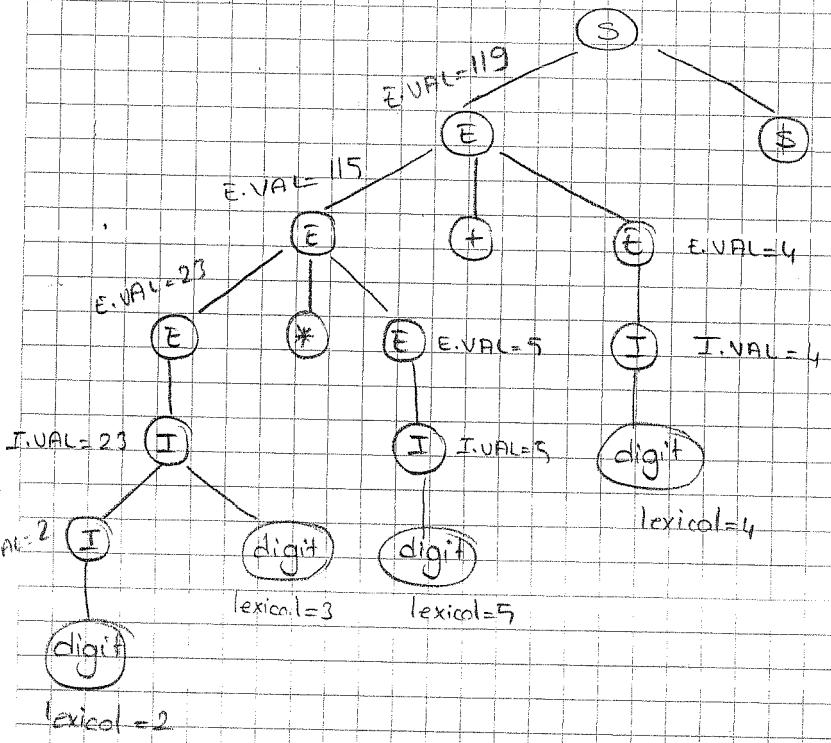
Örneğin 23 * 5 + 4 $ ise çıkış 119

Production

S ⇒ E $
E ⇒ E + E
E ⇒ E * E
E ⇒ (E) → hesapsal üstünlük (öncelik için)
E ⇒ I
I ⇒ I digit → bir rakamdan fazla olan
I ⇒ digit → karakter belirlemek } karakterler için

Semantik Hareketler

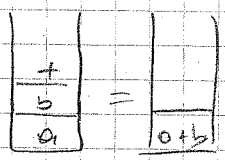
print E.VAL
E.VAL = E(1).VAL + E(2).VAL
E.VAL = " * "
E.VAL = E(1).VAL
E.VAL = I.VAL
I.VAL = 10. I(1).VAL + LEXICAL
I.VAL = LEXICAL



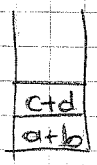
Postfix Notasyonu — Parantez kullanmadan işlem yapmayı sağlar

a+b ⟶ ab+
(a+b) * c ⟶ ab+c*
a*(b+c) ⟶ abc+*
(a+b) * (c+d) ⟶ ab+cd+*

* Yığıtlar kullanarak gerçekleştirilir

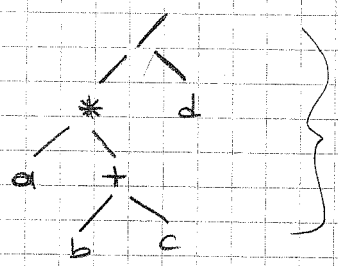
* 

Hangi operatörle karşılaşınca önceki iki karaktere uygulanır



Syntax Tree

a* (b+c) / d



Parse Tree

X

* Bir ifadenin parse ağacını çizebilmemiz için o ifadenin dil bilgisini bilmemiz gerekir. Gramer verilmediği için parse ağacını çizemeyiz. Fakat rastgele kurallar oluşturursak çizebiliriz.

Ödev : Verilen hesap makinasının semantik hareketler biçiminde kodlamasını yap. (Gerçekleştir. Dil farketmez (Java, C# ... vs vs))

19.03.14

Üç Adres Kodlaması : Orta seviye kodlama tekniğidir.

X+Y*Z ⇒ $T_1 = Y * Z$
$T_2 = X + T_1$

* if ve go to'lardan kaçta döngü ifadesi kullanılmaz

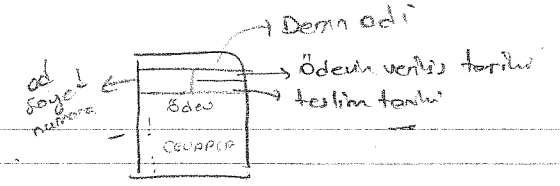
1) A = B op C
2) A = op B
3) go to L
4) if A relop B go to L
5) param A
   ⋮
   call P, n
6) A = B[I] , A[I] = B
7) A = adr B, A = *B , *A = B

ÖRN
A = -B * (C+D) gibi bir ifademiz var ise adres olarak;

$T_1 = -B$
$T_2 = C + D$
$T_3 = T_1 * T_2$
$A = T_3$ şeklinde yazılır.

ÖRN
if A<B then 1
else 0

(1) if A<B go to (4)
(2) T = 0
(3) go to (5)
(4) T = 1
(5) ...



ÖRNEK while(A<B) do

if(C<D) then X=Y+Z

Çözüm

```
100 : if (A<B) go to 102
101 : go to 107
102 : if (C<D) go to 104
103 : go to 100
104 : T = Y+Z
105 : X = T
106 : go to 100
107 : _ _
```

ÖDEV 100

ÖRNEK

for I=1 step 1 until N Üç adres kodu

do A[I]=0

Kod Optimizasyonu

Optimizasyonda döngüdeki satırların azaltılabilmesi önemli. Optimizasyon gerçekleştirilirken döngülerle daha çok göz önüne alınır.

∝ Döngüleri bulabilmek için go to'lara bakarız.

```
PROD = 0
I = 1
T1 = 4 * I
T2 = addr(A) - 4
T3 = T2[T1]
T4 = addr(B) - 4
T5 = T4[T1]
T6 = T3 * T5
PROD = PROD + T6
I = I + 1
if I ≤ 20 go to (3)
```

Temel Blokların Belirlenmesi

(1) ⇒ * Liderleri belirle

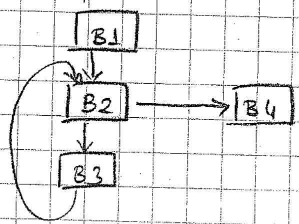
* İlk durum liderdir
* Şartlı veya şartsız bütün go to'ların hedefi olan satır liderdir.
* Şartlı go to'dan sonraki satır liderdir.

2) Bir liderden diğer lidere kadar olan satırlar temel blokları oluşturur.

ÖRNEK

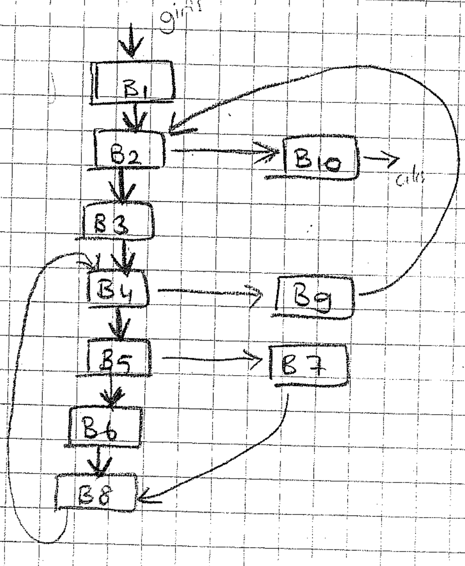
```
→ S1 :  read L          B1
  S2 :  n=0
  S3 :  k=0
  S4 :  m=1
→ S5 :  k=k+m           B2
  S6 :  C=k>1
  S7 :  if(c) go to S11
→ S8 :  n=n+1           B3
  S9 :  m=m+2
  S10:  go to S5
→ S11:  write n         B4
```

Kontrol akış grafı



ÖRNEK

```
→       i=1                      B1
→ L1 :  if(i>n) go to L6         B2
→       t1=0                     B3
        j=1
→ L2 :  if(j>n) go to L3         B4
→       tmp= tR - ts             B5
        if(tmp<0) go to L3
→       t1=t1+t5                 B6
        go to L4
→ L3 :  t1=t1+tR                 B7
→ L4 :  j=j+1                    B8
        go to L2
→ L5 :  i=i+1                    B9
        go to L1
→ L6 :  t1=0                     B10
```



Kontrol akış grafı:

ÖRNEK

→ L1 : if(i>0) go to L4 B1

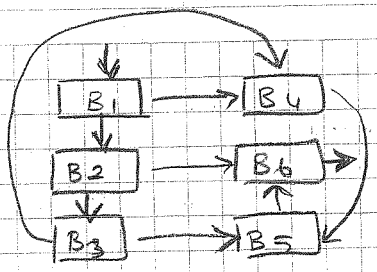
→ L2 : if(j>0) go to L6 B2

→ L3 : if(k>0) go to L5 B3

→ L4 : k=k-1 B4

→ L5 : j=j-1 B5

→ L6 : i=i-1 B6



ÖRNEK

→ L1 : i=0
L2 : go to L9 } B1

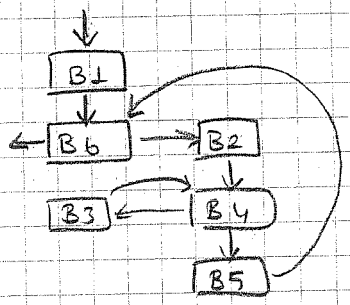
→ L3 : j=0
L4 : go to L7 } B2

→ L5 : s=s+j
L6 : j=j+1 } B3

→ L7 : if(j<n) go to L5 B4

→ L8 : i=i+1 B5

→ L9 : if i<n go to L3 B6



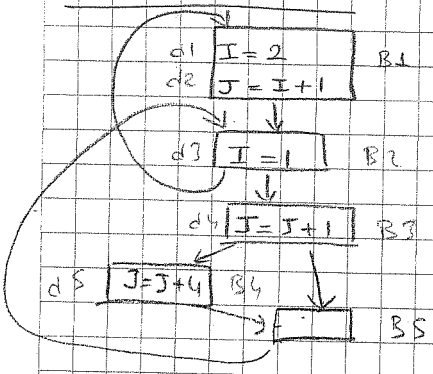
ÖDEV 2 — Bubble Sort, Quick Sort } Kontrol akış graflarını çiz.

✱ Önceki hafta kod optimizasyonu işlenmiş..

✱ Haftaya ödev raporunu unutmadd
✱ Ondan önceki haftanın da ödevini yapb

02.04.14

Veri Akış Analizi



Bloklar gösterir (B1, B2 gibi)

GEN [B]
KILL [B]
IN [B] = ∪ OUT [B]
OUT [B] = (IN [B] − KILL [B]) ∪ GEN [B]

GEN : B bloğu içindeki bütün tanımlamaların oluşturduğu kümedir.

KILL : " " " tanımlamaların dışında kalan tanımlamaların kümesidir.

IN : Bir bloğun girişlerini toplamıdır.

OUT :

Blok	GEN[B]	Bit vektör	KILL[B]	Bit Vektör
B1	{d1, d2}	11000	{d3, d4, d5}	00111
B2	{d3}	00100	{d1}	10000
B3	{d4}	00010	{d2, d5}	01001
B4	{d5}	00001	{d2, d4}	01010
B5	∅	00000 (d1 d2 d3 d4 d5)	∅	00000

OUT [B1] = GEN [B2] − KILL [B1] + GEN [B1]
= 00100 − 00111 + 11000 = 11000

IN [B2] = OUT [B1] + OUT [B5] = 11000 + 00000 = 11000

OUT [B2] = 11000 − 10000 + 00100 = 01100

IN [B3] = OUT [B2] = 01100

OUT [B3] = 01100 − 01001 + 00010 = 00110

IN [B4] = OUT [B3] = 00110

OUT [B4] = 00110 − 01010 + 00001 = 00101

IN [B5] = OUT [B3] + OUT [B4] = 00110 + 00101 = 00111

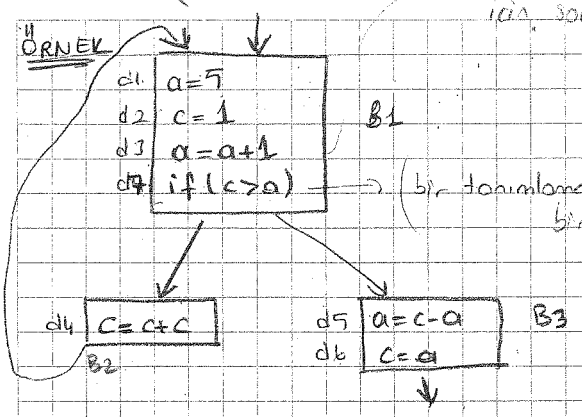
OUT [B5] = 00111 − 00000 + 00000 = 00111

(*) KILL'de örneğin GEN[B1]'de d2 ve d3 var. KILL[B1]'de bu ikisi dışında a ve c içeren tüm d'leri alıyoruz

GEN[B1]'de, d1 i almamamızın sebebi d1'de ani değer değiştiği için sonraki satırdakini alıyoruz.

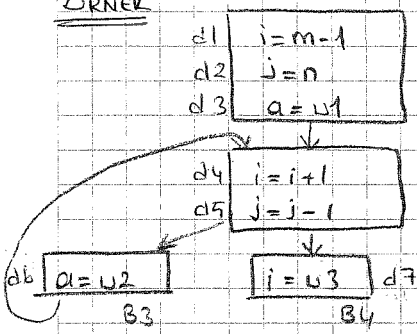
Kontrol akış grafiği için GEN ve KILL tablosunu bit vektörleriyle beraber veriniz.

(bir tanımlama değil bir koşul dahil değildir)



Blok	GEN[B]	Bit Vektör	KILL[B]	Bit Vektör
B1	{d2, d3}	011000	{d4, d5, d6}	000111
B2	{d4}	000100	{d2, d6}	010001
B3	{d5, d6}	000011	{d1, d2, d3, d4}	111100

ÖRNEK



Kontrol akış grafiği için GEN ve KILL tablosu ile başlangıç değerlerini 0 alarak out'ları hesaplayın.

Blok	GEN[B]	KILL[B]
B1	{d1, d2, d3}	{d4, d5, d6, d7}
B2	{d4, d5}	{d1, d2, d7}
B3	{d6}	{d3}
B4	{d7}	{d1, d4}

Bit vektörle

GEN

d1	d2	d3	d4	d5	d6	d7
1	1	1	0	0	0	0
0	0	0	1	1	0	0
0	0	0	0	0	1	0
0	0	0	0	0	0	1

KILL

0	0	0	1	1	1	1
1	1	0	0	0	0	1
0	0	1	0	0	0	0
1	0	0	1	0	0	0

(30 Nisan)
Vizeden sonraki hafta

4. Ödev Önceki sorunun IN, OUT'larını 3 iterasyon boyunca bulmak

İlk önce bulup, bulduğumuz değerlerle 3 iterasyon devam ettiriyoruz.

İlk IN'leri 0 aldık

ÖRNEK

Olusan hatalar: 4 yok

```
1.  class Error{
3.      private static final int x=4...;
4.      public static void main (String[] args){
6.          int[] a = new int[10];
7.          int i;
8.          boolean b;
9.          String s= "abc
10                    def";
12          x=1;
13          y=0;
14          i=a[10];
15          a[10]=1;
16          b=false;
17          if(b) a[i]=5;
        }}
```

lexical analizde hata verir ama yoğun syntax

→ syntax hata

→ yukarıda final değer atadığı için değer atayamam → semantik hata

→ semantik

Semantik hata

ÖRNEK

```
x=1
y=2
if(x<y) jmp L1;
jmp L2;
L1:...
```

Bu kod optimize edilebilir mi? Edilebiliyorsa nasıl edilir?

Bu satır gereksiz.

Örnek

```
a = (b+c)+m
x = b+c
y = (b+c)+z
```

```
x = b+c
a = x+m
y = x+z
```

Optimize edilebilir mi?

ÖRNEK
```
for(i=1 to 20)
    for(j=1 to 2)
        write(x[i,j])
```
Optimize edilebilir mi?

J. for'un altına 2 tane write yazarak optimize edebiliriz.

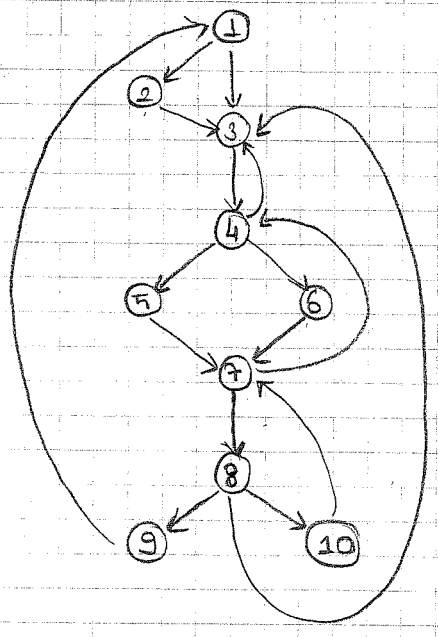
ÖRNEK
```
for k=1 to 1000
    c[k] = 2 * (p-q)*(n-k+1) / (sqrt(n)+n);
```
Optimize edilebilir mi?

```
t = 2*(p-q) / (sqrt(n)+n);
for k=1 to 1000
    c[k] = t*(n-k+1)
```
} 1000 kez matematiksel işlem yaptırmaktan kurtulmuş oluyoruz.

09.04.14

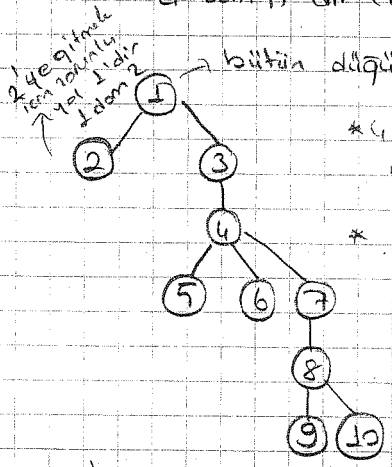
* Veri akış analizinin amacı optimizasyon ve tüm tanımlamalar için iterasyonlar ile optimizasyon yapmaktır.

Dominatör Ağacı



Kontrol akış grafı

* Döngüleri belirlemek için kullanılır.
* Dominatör = başlangıç düğümünden geçen her yol n düğümüne d'den geçerek gidiyorsa d Dom n dir. (d dom n'ye dominant)

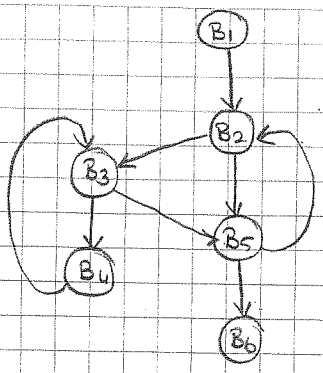


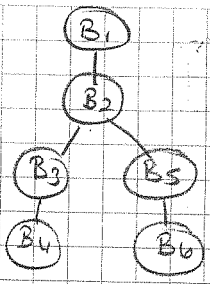
* 4 için 3'ten gelme mi gerekiyor
* Sağ sol önemli değil (değerlendirme)

Dominatör Ağacının.

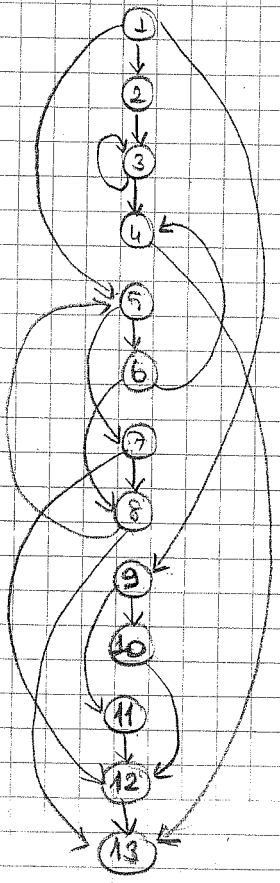
1. Kontrol Akış Grafı — Dominatör Ağacı

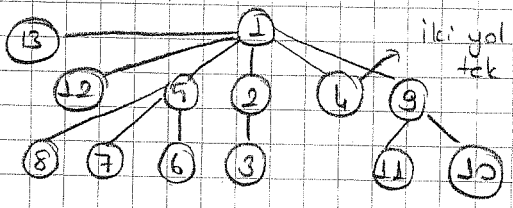
II. ÖRNEK





I. ÖRNEK

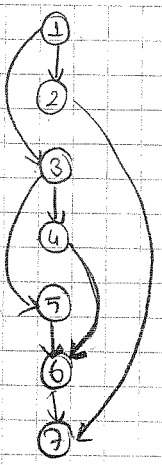


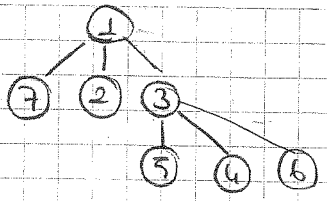


İki yol var bunu tek e indirmek için tek düğüme gidiyor.

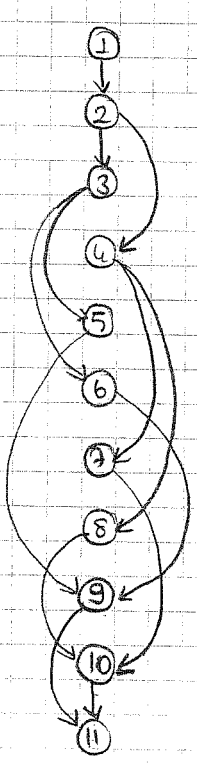
* Birden fazla yol varsa o yolların sonlarına bakıp hepsinin ortak geçtiği durumu alıyoruz.

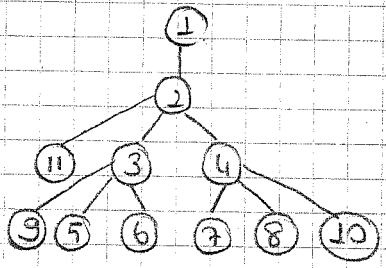
"ÖRNEK





"ÖRNEK





II ÖRNEK