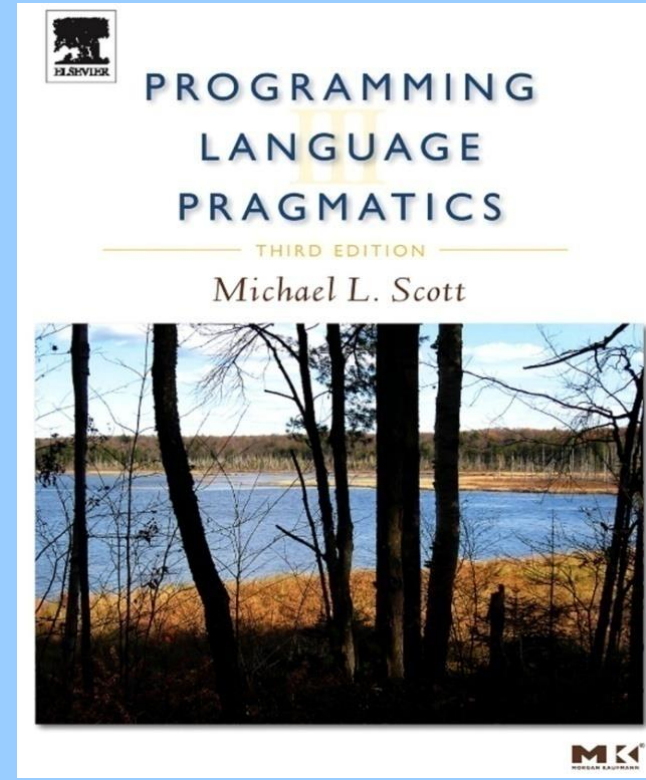
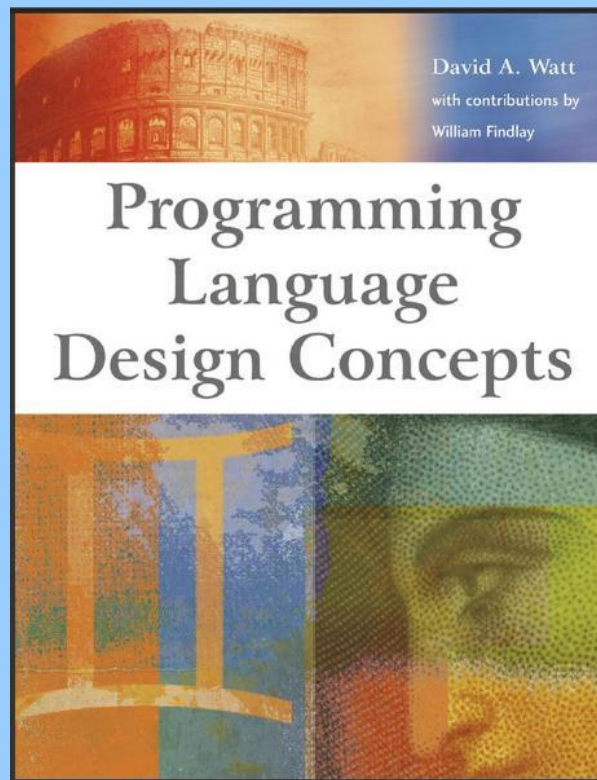
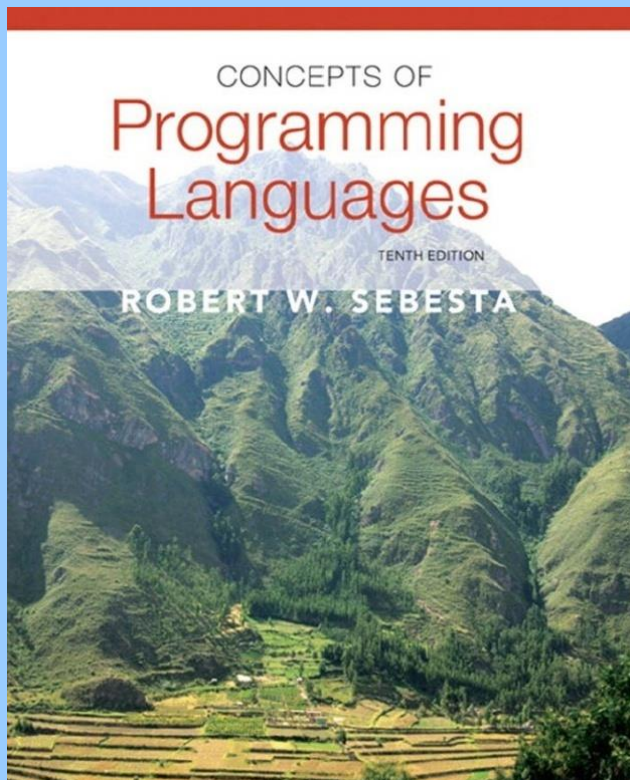


# Bölüm 4: Sözcüksel (Lexical) ve Sentaks Analiz

---



# Bölüm 4 Konular

---

1. Giriş
2. Sözcüksel Analiz (Lexical Analysis)
3. Ayırıştırma (Parsing) Problemi
4. Özyineli–Azalan Ayırıştırma (Recursive–Descent Parsing)
5. Aşağıdan–Yukarıya Ayırıştırma (Bottom–Up Parsing)

## 4.1 Giriş

---

- Dil implementasyon sistemleri, belirli implementasyon yaklaşımına aldirmeden kaynak kodu analiz etmelidir
- Hemen hemen bütün sentaks analizi kaynak dilin sentaksının biçimsel tanımlamasına dayalıdır (BNF)

## 4.1 Giriş (Devamı)

---

- Bir dil işlemcisinin sentaks analizi bölümü genellikle iki kısımdan oluşur:
  - Bir düşük–düzeyli kısım, **sözcüksel analizci (lexical analyzer)** (matematiksel olarak, düzenli bir gramere dayalı bir sonlu otomatı (finite automaton))
  - Bir yüksek–düzeyli (high–level) kısım, **sentaks analizci (syntax analyzer)**, veya ayrıştırıcı (parser) (matematiksel olarak, içerik–bağımsız gramere dayalı bir aşağı–itme otomatı (push–down automaton), veya BNF)

## 4.1 Giriş (Devamı)

---

- Sentaksı tanımlamak için BNF kullanmanın nedenleri :
  - Net ve özlü bir sentaks tanımı sağlar
  - Ayırıştırıcı doğrudan BNF ye dayalı olabilir
  - BNF ye dayalı ayırıştırıcıların bakımı daha kolaydır

## 4.1 Giriş (Devamı)

---

- Sözcüksel (lexical) ve sentaks (syntax) analizini ayırmanın nedenleri:
  - **Basitlik** – sözcüksel analiz (lexical analysis) için daha az karmaşık yaklaşımlar kullanılabilir; bunları ayırmak ayrıştırıcıyı basitleştirir
  - **Verimlilik** – ayırmak sözcüksel analizcinin (lexical analyzer) optimizasyonuna imkan verir (sentaks analizciyi optimize etmek sonuç vermez, verimli değil)
  - **Taşınabilirlik** – sözcüksel analizcinin (lexical analyzer) bölümleri taşınabilir olmayabilir, fakat ayrıştırıcı her zaman taşınabildirir

## 4.2 Sözcüksel (Lexical) Analiz

---

- Sözcüksel analizci (lexical analyzer), karakter stringleri için desen eşleştiricidir
- Sözcüksel analizci ayrıştırıcı için bir “ön-uç”tur (“front-end”)
- Kaynak programın birbirine ait olan altstringlerini tanımlar – **lexeme’ler**
  - **Lexeme**ler, **jeton** (token) adı verilen sözcüksel (lexical) bir kategoriyle ilişkilendirilmiş olan bir karakter desenini eşleştirir
  - **sum** bir **lexeme**dir; jetonu (token) **IDENT** olabilir

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

- Sözcüksel analizci (lexical analyzer), genellikle ayrıştırıcının sonraki jetona (token) ihtiyaç duyduğunda çağırdığı fonksiyondur. Sözcüksel analizci (lexical analyzer) oluşturmaya üç yaklaşım:
  - Jetonların biçimsel tanımı yazılır ve bu tanıma göre tablo-sürümlü sözcüksel analizciyi oluşturan yazılım aracı kullanılır
  - Jetonları tanımlayan bir durum diyagramı tasarlanır ve durum diyagramını implement eden bir program yazılır
  - Jetonları tanımlayan bir durum diyagramı tasarlanır ve el ile durum diyagramının tablo-sürümlü bir implementasyonu yapılır
- Sadece ikinci yaklaşımdan bahsedeceğiz



## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

- Durum diyagramı tasarımı:
  - Saf (Naive) bir durum diyagramı kaynak dildeki her karakterde, her durumdan bir geçişe sahip olacaktı – böyle bir diyagram çok büyük olurdu!

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

- Çoğu kez, durum diyagramı basitleştirmek için geçişler birleştirilebilir
  - Bir tanıtcıyı (identifier) tanırken, bütün büyük (uppercase) ve küçük (lowercase) harfler eşittir
    - Bütün harfleri içeren bir karakter sınıfı (character class) kullanılır
  - Bir sabit tamsayıyı (integer literal) tanırken, bütün rakamlar (digits) eşittir – bir rakam sınıfı (digit class) kullanılır

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

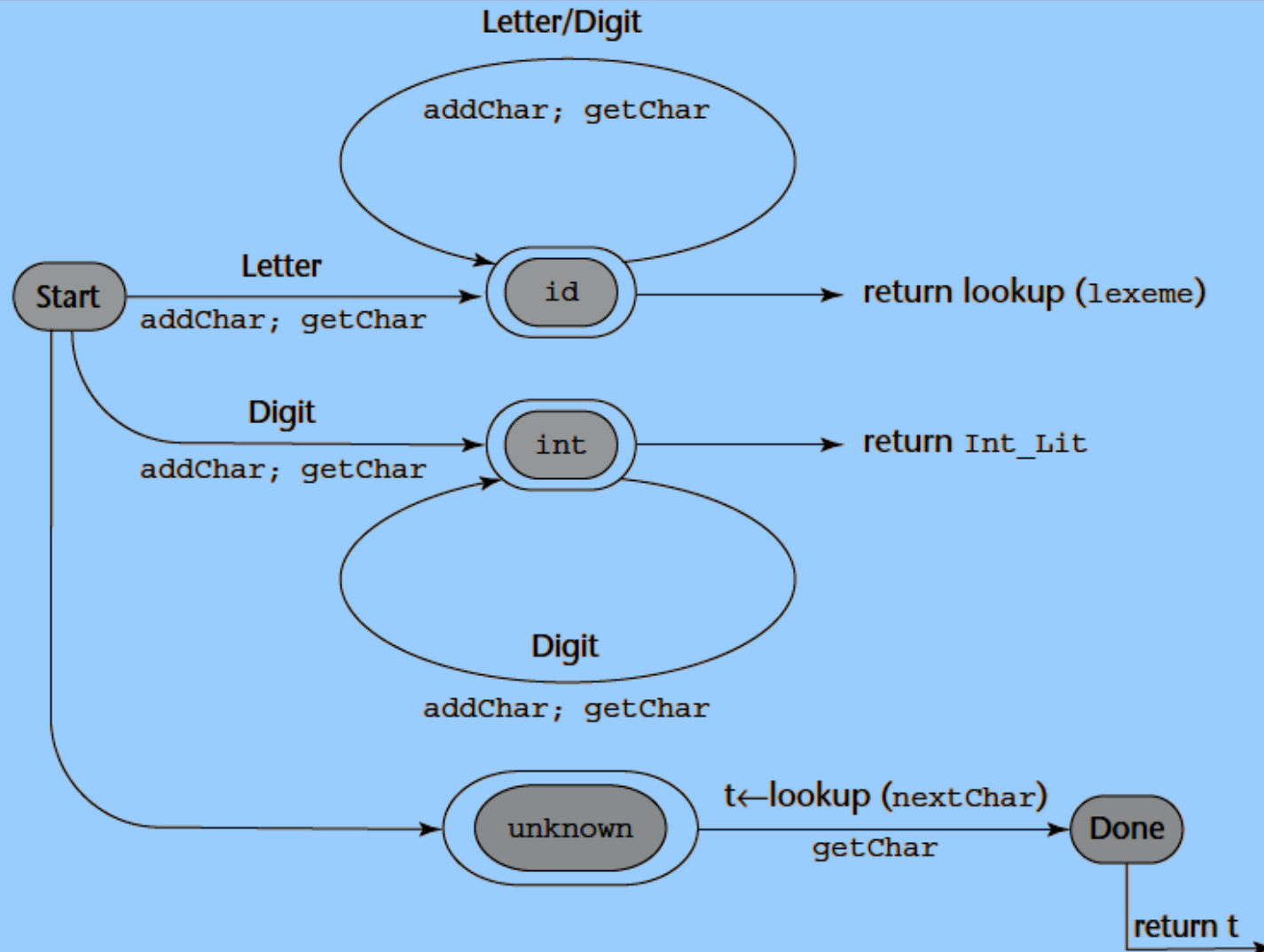
- Ayrılmış sözcükler (reserved words) ve tanıtıcılar (identifiers) birlikte tanınabilir (her bir ayrılmış sözcük için programın bir parçasını almak yerine)
  - Olası bir tanıtıcının (identifier) aslında ayrılmış sözcük olup olmadığına karar vermek için, tabloya başvurma (table lookup) kullanılır

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

- Kullanışlı yardımcı altprogramlar:
  - `getChar` – girdinin sonraki karakterini alır, bunu `nextChar` içine koyar, sınıfını belirler ve sınıfı `charClass` içine koyar
  - `addChar` – `nextChar` dan gelen karakteri **lexeme**nin biriktirildiği yere koyar (**lexeme** dizisinin sonuna ekler)
  - Arama (lookup) – **lexeme** deki stringin ayrılmış sözcük (reserved word) olup olmadığını belirler ve onun kodunu döndürür

# Durum Diyagramı (State Diagram)



Adları, parantezleri ve aritmetik operatörleri tanıyan bir durum diyagramı

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

implementasyon (başlatma varsayalım):

```
int lex() {
    getChar();
    switch (charClass) {
        case LETTER:
            addChar();
            getChar();
            while (charClass == LETTER || charClass == DIGIT)
            {
                addChar();
                getChar();
            }
            return lookup(lexeme);
            break;
        ...
    }
}
```

## 4.2 Sözcüksel (Lexical) Analiz (Devamı)

---

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

# Sözcüksel (Lexical) Analiz

---

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



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



## 4.3 Ayırıştırma (Parsing) Problemi

---

- Ayırıştırıcının amaçları, bir girdi programı verildiğinde:
  - Bütün sentaks hatalarını bulur; her birisi için, uygun bir tanılayıcı (iyileştirici) mesaj üretir, ve gerekirse düzeltmeler yapar
  - Ayırıştırma ağacını üretir, veya en azından program için ayırıştırma ağacının izini (dökümünü) üretir

## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

---

- Ayırıştırıcıların iki kategorisi:
  - Yukarıdan-aşağıya (Top down) – ayırıştırma ağacını kökten başlayarak oluşturur
    - Ayırıştırma ağacını **preorder**da izler veya oluşturur
  - Aşağıdan-yukarıya (Bottom up) – ayırıştırma ağacını, yapraklardan başlayarak oluşturur
- Ayırıştırıcılar, girdide sadece bir jeton (token) ileriye bakar

## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

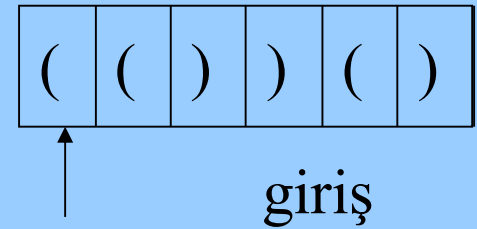
---

- Yukarıdan-aşağıya ayırıştırıcılar (Top-down parsers)
  - Bir  $xA\alpha$  sağ cümlesel formu (right sentential form) verildiğinde , ayırıştırıcı, sadece A'nın ürettiği ilk jetonu (token) kullanarak, ensol türevdeki (leftmost derivation) sonraki cümlesel formu (sentential form ) elde etmek için doğru olan A-kuralını (A-rule) seçmelidir
- En yaygın yukarıdan-aşağıya ayırıştırma (top-down parsing) algoritmaları:
  - Özyineli azalan (recursive-descent)- kodlanmış bir implementasyon
  - LL ayırıştırıcılar (parser) - tablo sürümlü implementasyon

# $((()))$ için Leftmost Türetme

---

$L$	$// L \Rightarrow (L)L$
$\Rightarrow (L)L$	$// L \Rightarrow (L)L$
$\Rightarrow ((L)L)L$	$// L \Rightarrow \varepsilon$
$\Rightarrow (()L)L$	$// L \Rightarrow \varepsilon$
$\Rightarrow (())L$	$// L \Rightarrow \varepsilon$
$\Rightarrow (())(L)L$	$// L \Rightarrow (L)L$
$\Rightarrow (())()L$	$// L \Rightarrow \varepsilon$
$\Rightarrow (())()$	



## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

Yukarıdan-aşağıya ayırıştırıcılar (Top-down parsers)

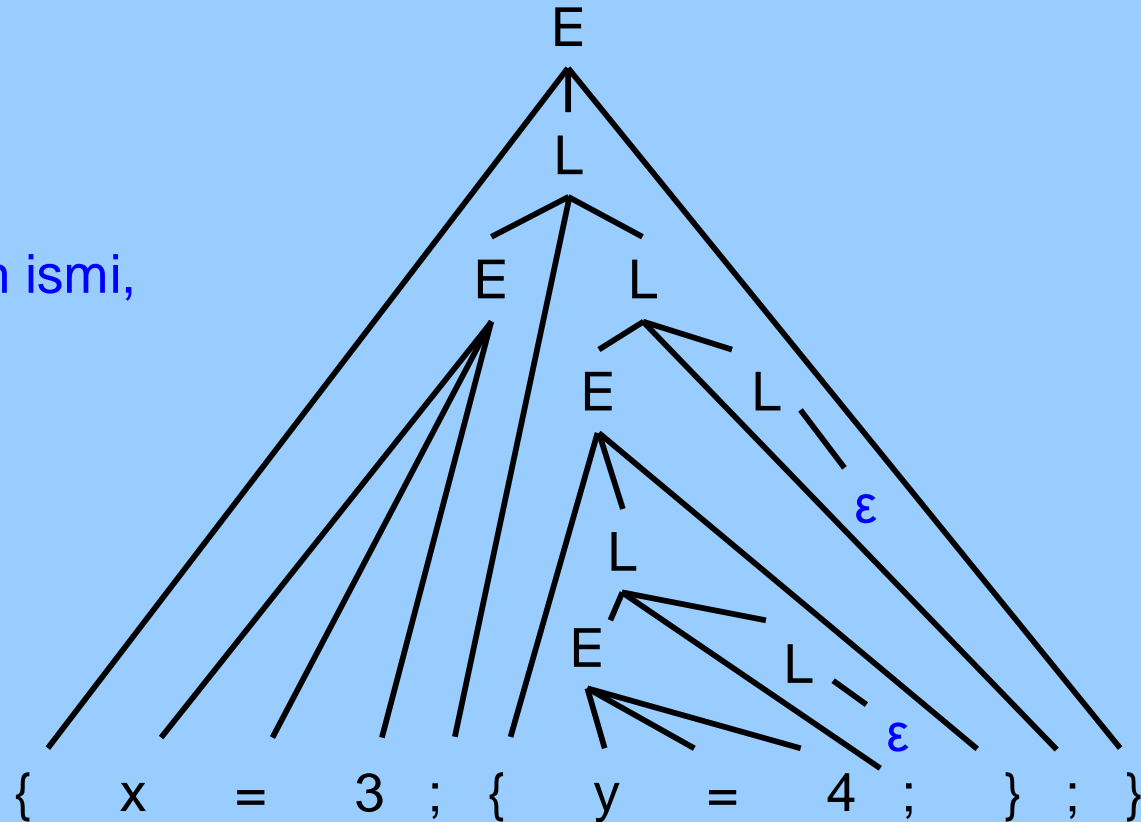
$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \epsilon$

(Varsay: id - değişken ismi,  
n - tamsayı)

$\{ x = 3 ; \{ y = 4 ; \} ; \}$

için ayırıştırma  
ağacını göster



## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

---

- Aşağıdan–yukarıya ayırıştırıcılar (bottom–up parsers)
  - Bir  $\alpha$  sağ cümlesel formu (right sentential form) verildiğinde,  $\alpha$  nın sağ türevde önceki cümlesel formu üretmesi için azaltılması gerekli olan, gramerde kuralın sağ tarafında olan altstringinin ne olduğuna karar verir
  - En yaygın aşağıdan–yukarıya ayırıştırma algoritmaları LR ailesindedir

## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

Aşağıdan-yukarıya ayırıştırıcılar (bottom-up parsers)

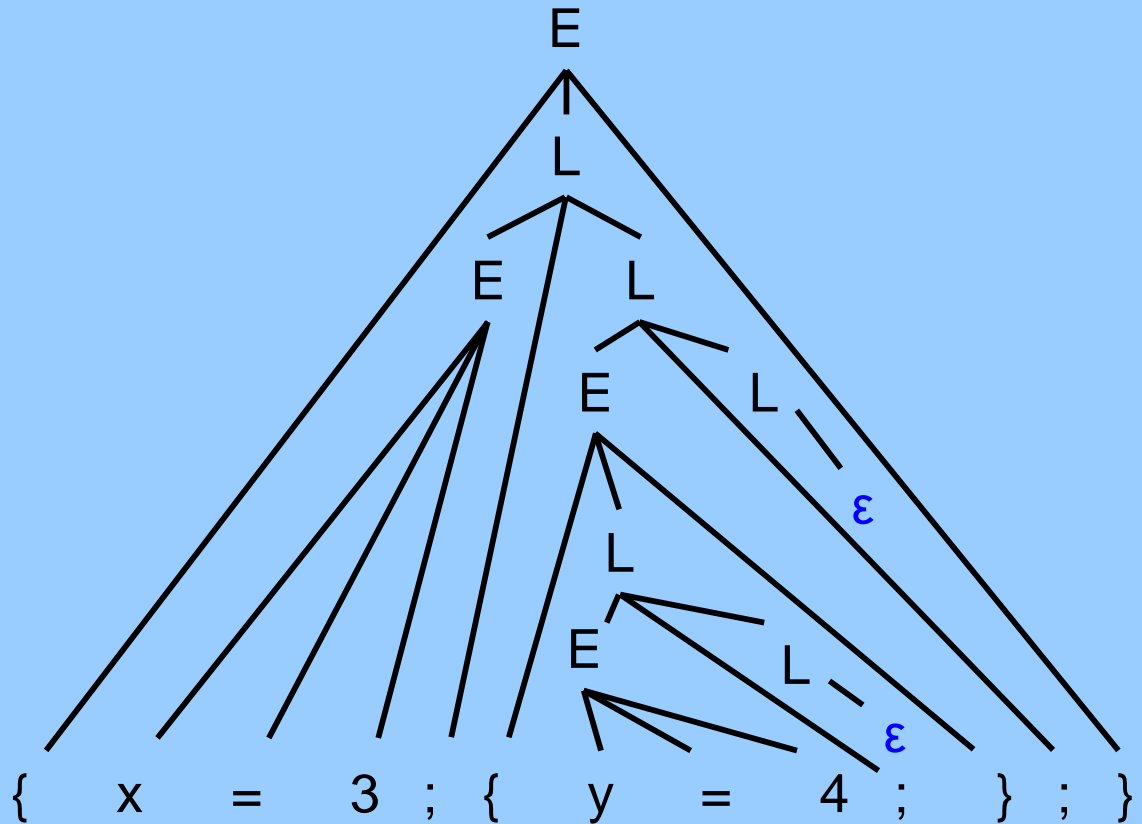
$E \rightarrow id = n \mid \{ L \}$

$L \rightarrow E ; L \mid \epsilon$

$\{ x = 3 ; \{ y = 4 ; \} ; \}$

için ayırıştırma  
ağacını göster

Oluşturulan sonuç ağaçlar  
yukarıdan-aşağıya ile  
aynıdır; sadece düğümlerin  
ağaca eklenme sırası  
değişiktir



## 4.3 Ayırıştırma (Parsing) Problemi (Devamı)

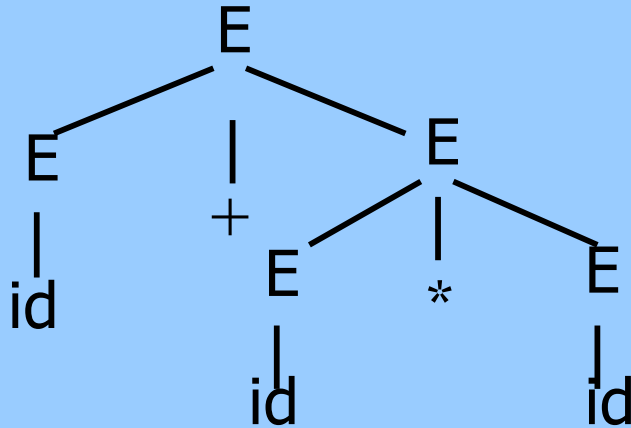
---

- Ayırıştırmanın Karmaşıklığı
  - Herhangi bir belirsiz–olmayan gramer için çalışan ayırıştırıcılar, karmaşık ve etkisizdir ( $O(n^3)$ ,  $n$  girdinin uzunluğu)
  - Derleyiciler, sadece bütün belirsiz–olmayan gramerlerin bir altkümesi için çalışan ayırıştırıcıları kullanır, fakat bunu lineer zamanda yapar ( $O(n)$ ,  $n$  girdinin uzunluğu)



# Ayrıştırma Ağaçları ve Türetmeler

---



Top-down ayrıştırma

$E \Rightarrow E + E$   
 $\Rightarrow id + E$   
 $\Rightarrow id + E * E$   
 $\Rightarrow id + id * E$   
 $\Rightarrow id + id * id$

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)

---

- Özyineli–azalan işlem (Recursive–descent Process) (Yukarıdan–Aşağıya ayırıştırma yapar)
  - Gramerde her bir nonterminal için o nonterminal tarafından üretilebilen cümleleri ayırıştırabilen bir altprogram vardır
  - EBNF, özyineli–azalan ayırıştırıcıya (recursive–descent parser) temel oluşturmak için idealdir, çünkü EBNF nonterminal sayısını minimize eder

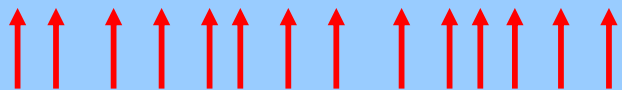
## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)

---

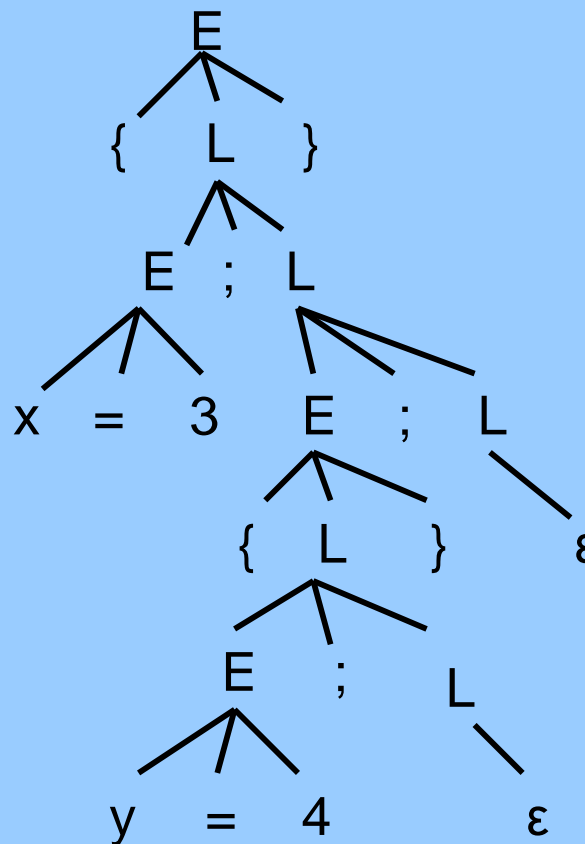
$E \rightarrow \text{id} = n \mid \{ L \}$

$L \rightarrow E ; L \mid \epsilon$

$\{ x = 3 ; \{ y = 4 ; \} ; \}$



lookahead



## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing) (Devamı)

---

- Basit deyimler (expressions) için bir gramer:

`<expr> → <term> { (+ | -) <term> }`

`<term> → <factor> { (* | /) <factor> }`

`<factor> → id | ( <expr> )`

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing) (Devamı)

---

- **Lex** isimli, sonraki jeton kodunu **nextToken** içine koyan bir sözlüksel analizci (lexical analyzer) olduğunu varsayalım
- Sadece bir sağdaki kısım (RHS) olduğunda kodlama işlemi:
  - Sağdaki kısımda (RHS) olan her bir terminal sembol için, onu bir sonraki girdi jetonuyla karşılaştır; eğer eşleşiyorsa, devam et; değilse hata vardır
  - Sağdaki kısımda (RHS) her bir nonterminal sembol için, onunla ilgili ayırıştırıcı alt programını çağırır

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

```
/* expr fonksiyonu
   Dilde kural tarafından üretilen
   stringleri ayırıştırır:
   <expr> → <term> { (+ | -) <term> }
*/

void expr() {

/* ilk terimi ayırıştır*/

term();

...
}
```

## 4.4 Özyineli–azalan Ayırıştırma(Recursive–Descent Parsing)(Devamı)

---

```
/* Sonraki jeton(token) + veya - olduğu sürece, sonraki  
   jetonu(token) almak için lex'i çağır, ve sonraki terimi  
   ayırıştır  
*/
```

```
while (nextToken == PLUS_CODE ||  
       nextToken == MINUS_CODE) {  
    lex();  
    term();  
}  
}
```

- Bu özel rutin hataları bulmaz
- Kural: Her ayırıştırma rutini sonraki jetonu **nextToken**'da bırakır

## 4.4 Özyineli–azalan Ayırıştırma (Recursive– Descent Parsing)(Devamı)

---

- Birden fazla sağdaki kısmı (RHS) olan bir nonterminal, hangi sağdaki kısmı (RHS) ayrıştıracağına karar vermek için bir başlangıç işlemine gerek duyar
  - Doğru sağdaki kısım (RHS), girdinin sonraki jetonunu temel alarak seçilir (lookahead)
  - Bir eşlenik bulana kadar sonraki jeton her bir sağdaki kısım (RHS) tarafından üretilebilen ilk jetonla karşılaştırılır
  - Eğer eşlenik bulunmazsa, bu bir sentaks hatasıdır.



## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

```
/* factor fonksiyonu dilde şu kuralın  
   ürettiği stringleri ayırıştırır:  
   <factor> -> id | (<expr>) */
```

```
void factor() {
```

```
/* Hangi RHS olduğunu belirle*/
```

```
if (nextToken) == ID_CODE)
```

```
/* RHS id si için, lex 'i çağır*/
```

```
lex();
```

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

```
/* Eğer sağdaki kısım(RHS) (<expr>) ise - sol  
   parantezi ihmal ederek lex'i çağır, expr'yi çağır,  
   ve sağ parantezi kontrol et */  
  
else if (nextToken == LEFT_PAREN_CODE) {  
    lex();  
    expr();  
    if (nextToken == RIGHT_PAREN_CODE)  
        lex();  
    else  
        error();  
} /* End of else if (nextToken == ... */  
  
else error(); /* Hiçbir RHS eşleşmedi*/  
}
```

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

- LL Gramer Sınıfı (LL Grammar Class)
  - Sol Özyineleme (Left Recursion) Problemi
    - Eğer bir gramerin sol özyinelemesi varsa, doğrudan veya dolaylı, yukarıdan–aşağıya (Top–down) ayırıştırıcının temeli olamaz
      - Bir gramer, sol özyinelemeyi yok etmek için değiştirilebilir

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

- Yukarıdan–aşağıya ayırştırmaya izin vermeyen gramerlerin diğer bir özelliği pairwise disjointness (**çiftli ayrıklık**) eksikliğidir
    - Doğru olan sağ kısmı (RHS) **lookahead**ın bir jetonuna dayanarak belirleyememesi
    - Tanım:  $FIRST(\alpha) = \{a \mid \alpha \Rightarrow^* a\beta\}$   
(Eğer  $\alpha \Rightarrow^* \varepsilon$  ise,  $\varepsilon \in FIRST(\alpha)$  içindedir)
- (**First(X)** herhangi bir cümlesel formda X'ten türetilen ilk terminal kümesidir)
- $\alpha \Rightarrow^*$  : sıfır veya daha fazla türetme varsa

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

- **Pairwise Disjointness Testi:**
  - Her bir nonterminal  $A$  için, birden fazla sağ kısmı (RHS) olan gramerde, her bir kural çifti  $A \rightarrow \alpha_i$  ve  $A \rightarrow \alpha_j$  için, şu doğru olmalıdır:
$$\text{FIRST}(\alpha_i) \cap \text{FIRST}(\alpha_j) = \phi$$
- Türetmede üretilecek ilk terminal sembol tek olmalıdır.
- Örnekler:

$A \rightarrow a \mid bB \mid cAb$

$A \rightarrow a \mid aB$       disjoint değil!

## 4.4 Özyineli–azalan Ayırıştırma (Recursive–Descent Parsing)(Devamı)

---

- Sol çarpan alma (Left factoring) problemi çözebilir  
Şu ifadeyi:

$\langle \text{variable} \rangle \rightarrow \text{identifier} \mid \text{identifier} [\langle \text{expression} \rangle]$

aşağıdakilerden biriyle değiştirin:

$\langle \text{variable} \rangle \rightarrow \text{identifier} \langle \text{new} \rangle$

$\langle \text{new} \rangle \rightarrow \varepsilon \mid [\langle \text{expression} \rangle]$

veya

$\langle \text{variable} \rangle \rightarrow \text{identifier} [[\langle \text{expression} \rangle]]$

(dıştaki köşeli parantezler EBNF'nin metasembolleridir)

# Özyineli–azalan Ayırıştırma Problemler

---

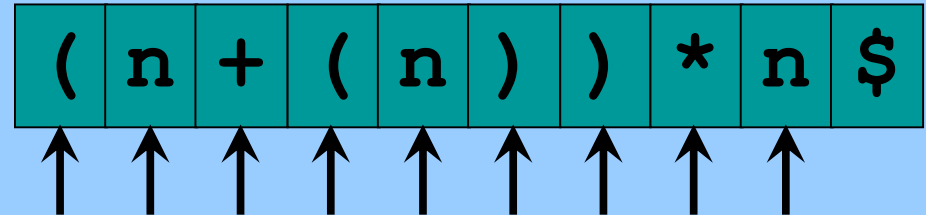
- Gramerleri EBNF'ye dönüştürmek zor
- Her noktada, hangi kuralın (production) kullanılacağına karar veremez
- $\lambda$ -production  $A \rightarrow \lambda$  ne zaman kullanacağına karar veremez

# LL(1) Ayırıştırma Örneği

$E \Rightarrow TX$   
 $\Rightarrow FNX$   
 $\Rightarrow (E) NX$   
 $\Rightarrow (TX) NX$   
 $\Rightarrow (FNX) NX$   
 $\Rightarrow (nNX) NX$   
 $\Rightarrow (nX) NX$   
 $\Rightarrow (nATX) NX$   
 $\Rightarrow (n+TX) NX$   
 $\Rightarrow (n+FNX) NX$   
 $\Rightarrow (n+(E)NX) NX$   
 $\Rightarrow (n+(TX)NX) NX$   
 $\Rightarrow (n+(FNX)NX) NX$   
 $\Rightarrow (n+(nNX)NX) NX$   
 $\Rightarrow (n+(nX)NX) NX$   
 $\Rightarrow (n+(n)NX) NX$   
 $\Rightarrow (n+(n)X) NX$   
 $\Rightarrow (n+(n)) NX$   
 $\Rightarrow (n+(n)) MFNX$   
 $\Rightarrow (n+(n)) * FNX$   
 $\Rightarrow (n+(n)) * nNX$   
 $\Rightarrow (n+(n)) * nX$   
 $\Rightarrow (n+(n)) * n$

Bitti

n
N
X
)
N
*
n
N
X
\$



$E \rightarrow T X$   
 $X \rightarrow A T X \mid \lambda$   
 $A \rightarrow + \mid -$   
 $T \rightarrow F N$   
 $N \rightarrow M F N \mid \lambda$   
 $M \rightarrow *$   
 $F \rightarrow ( E ) \mid n$



# LL(1) Ayırıştırma Algoritması

---

Giriş symbolünü yığına at

WHILE yığın boş değil(\$ yığının tepesinde değil) ve token akışı boş değil (bir sonraki giriş tokenı \$ değil)

SWITCH (Yığının tepesi, sonraki token)

CASE (terminal  $a$ ,  $a$ ):

Pop yığın;      Sonraki tokenı al

CASE (nonterminal  $A$ , terminal  $a$ ):

IF ayırıştırma tablosu girişi  $M[A, a]$  boş değil THEN  
ayırıştırma tablosu girişi  $M[A, a]$  dan  $A \rightarrow X_1 X_2 \dots X_n$  al

Pop yığın;

$X_n \dots X_2 X_1$  'yi yığına bu sırada Push

ELSE Error

CASE ( $\$, \$$ ):      Accept

OTHER:              Error

# LL(1) Ayırıştırma Tablosu

---

Nonterminal  $N$  yığının  
*tepesinde* ve sonraki  
token  $t$  ise, hangi kural  
(production) kullanılacak?

- $N \rightarrow X$  kuralını seç öyleki
  - $X \Rightarrow^* tY$  ya da
  - $X \Rightarrow^* \lambda$  ve  $S \Rightarrow^* WNtY$

t	X
Y	t
Q	Y

t	...	...	...
---	-----	-----	-----



# First Kümesi

---

- $X, \lambda$  olsun ya da  $V$  veya  $T'$ 'de olsun.
- $\text{First}(X)$  herhangi bir cümlesel formda  $X$ 'ten türetilen ilk terminal kümesidir
  - $X$  bir terminal ya da  $\lambda$  ise, o zaman  $\text{First}(X) = \{X\}$ .
  - $X$  bir nonterminal ve  $X \rightarrow X_1 X_2 \dots X_n$  bir kural ise, o zaman
    - $\text{First}(X_1) - \{\lambda\}$   $\text{First}(X)$ 'in bir subsetidir
    - $\text{First}(X_j) - \{\lambda\}$   $\text{First}(X)$ 'in bir subsetidir, eğer tüm  $j < i$  için  $\text{First}(X_j)$   $\{\lambda\}$ 'yı içerirse
    - $\lambda$   $\text{First}(X)$ 'in içindedir, tüm  $j \leq n$  için  $\text{First}(X_j)$   $\{\lambda\}$ 'yı içerirse

# First Küme Örnekleri

---

$\text{exp} \rightarrow \text{exp addop term} \mid \text{term}$   
 $\text{addop} \rightarrow + \mid -$   
 $\text{term} \rightarrow \text{term mulop factor} \mid \text{factor}$   
 $\text{mulop} \rightarrow *$   
 $\text{factor} \rightarrow (\text{exp}) \mid \text{num}$   
 $\text{First}(\text{addop}) = \{+, -\}$   
 $\text{First}(\text{mulop}) = \{*\}$   
 $\text{First}(\text{factor}) = \{(\text{, num}\}$   
 $\text{First}(\text{term}) = \{(\text{, num}\}$   
 $\text{First}(\text{exp}) = \{(\text{, num}\}$

$\text{st} \rightarrow \text{ifst} \mid \text{other}$   
 $\text{ifst} \rightarrow \text{if ( exp ) st elsepart}$   
 $\text{elsepart} \rightarrow \text{else st} \mid \lambda$   
 $\text{exp} \rightarrow 0 \mid 1$

$\text{First}(\text{exp}) = \{0, 1\}$   
 $\text{First}(\text{elsepart}) = \{\text{else}, \lambda\}$   
 $\text{First}(\text{ifst}) = \{\text{if}\}$   
 $\text{First}(\text{st}) = \{\text{if}, \text{other}\}$

# First(A) Bulma Algoritması

For tüm a terminalleri,  $\text{First}(a) = \{a\}$   
For tüm A nonterminalleri,  $\text{First}(A) := \{\}$   
While herhangi  $\text{First}(A)$ 'ya değişiklik var  
    For her kural  $A \rightarrow X_1 X_2 \dots X_n$   
        For  $\{X_1, X_2, \dots, X_n\}$ 'de her  $X_i$   
            If for tüm  $j < i$   $\text{First}(X_j)$   $\lambda$  içerir,  
                Then  
                     $\text{First}(X_i) - \{\lambda\}$ 'yı  $\text{First}(A)$ 'ya ekle  
            If  $\lambda$ ;  $\text{First}(X_1), \text{First}(X_2), \dots$ , ve  
                 $\text{First}(X_n)$ 'de  
                    Then  $\lambda$ 'yı  $\text{First}(A)$ 'ya ekle

If A bir terminal ya da  $\lambda$ , then  $\text{First}(A) = \{A\}$ .  
If A bir nonterminal, then for her kural  $A \rightarrow X_1 X_2 \dots X_n$ ,  $\text{First}(A) - \{\lambda\}$ 'yi içerir.  
If aynı zamanda for bazı  $i < n$ ,  $\text{First}(X_1), \text{First}(X_2), \dots$ , ve  $\text{First}(X_i)$   $\lambda$ 'yı içerir, then  $\text{First}(A) - \{\lambda\}$ 'yi içerir.  
If  $\text{First}(X_1), \text{First}(X_2), \dots$ , ve  $\text{First}(X_n)$   $\lambda$ 'yı içerir, then  $\text{First}(A)$  da  $\lambda$  'yı içerir.

# First Küme Bulma: Bir Örnek

---

$\text{exp} \rightarrow \text{term exp}'$

$\text{exp}' \rightarrow \text{addop term exp}' \mid \lambda$

$\text{addop} \rightarrow + \mid -$

$\text{term} \rightarrow \text{factor term}'$

$\text{term}' \rightarrow \text{mulop factor term}'$   
 $\mid \lambda$

$\text{mulop} \rightarrow *$

$\text{factor} \rightarrow ( \text{exp} ) \mid \text{num}$

	First
exp	
exp'	$\lambda$
addop	+ -
term	( num
term'	$\lambda$
mulop	*
factor	( num

# Follow Kümesi

---

- \$ giriş tokenlarının sonunu gösterebilir
- A başlangıç sembolü ise, o zaman \$,  $\text{Follow}(A)$ 'dadır.
- Bir  $B \rightarrow X A Y$  kuralı varsa, o zaman  $\text{First}(Y) - \{\lambda\}$   $\text{Follow}(A)$ 'dadır.
- $B \rightarrow X A Y$  üretimi (kuralı) varsa ve  $\lambda$ ,  $\text{First}(Y)$ 'de ise, o zaman  $\text{Follow}(A)$ ,  $\text{Follow}(B)$ 'yi içerir.

# Follow(A) Bulma Algoritması

Follow(S) = {\$}

FOR V-{S}'de her A

Follow(A)={}

WHILE bazı Follow kümelerine değişiklik yapılır

FOR each production  $A \rightarrow X_1 X_2 \dots X_n$ ,

FOR each nonterminal  $X_i$

First( $X_{i+1} X_{i+2} \dots X_n$ ) - { $\lambda$ } 'yi  
Follow( $X_i$ )'e ekle.

(NOT: If  $i=n$ ,  $X_{i+1} X_{i+2} \dots X_n = \lambda$ )

IF  $\lambda$  First( $X_{i+1} X_{i+2} \dots X_n$ )'te ise THEN

Follow(A)'yı to Follow( $X_i$ )'e ekle

If A başlangıç  
sembolü, then \$  
Follow(A)'dadır.

If bir  $A \rightarrow Y X Z$   
kuralı varsa,  
then First(Z) - { $\lambda$ }  
Follow(X)'tedir.

If  $B \rightarrow X A Y$   
production varsa  
ve  $\lambda$  First(Y)'de  
ise, then  
Follow(A)  
Follow(B)'yi  
içerir.



# Follow Kümesi Bulma: Bir Örnek

$\text{exp} \rightarrow \text{term exp}'$

$\text{exp}' \rightarrow \text{addop term exp}' \mid \lambda$

$\text{addop} \rightarrow + \mid -$

$\text{term} \rightarrow \text{factor term}'$

$\text{term}' \rightarrow \text{mulop factor term}'$   
 $\mid \lambda$

$\text{mulop} \rightarrow *$

$\text{factor} \rightarrow ( \text{exp} ) \mid \text{num}$

	First	Follow
exp	( num	\$ )
exp'	$\lambda$ + -	\$ )
addop	+ -	
term	( num	+ - \$ )
term'	$\lambda$ *	
mulop	*	
factor	( num	

# LL(1) Ayırıştırma Tabloları Oluşturma

---

```
FOR her nonterminal A ve bir  $A \rightarrow X$  productionı
  FOR First(X)'te her token
     $A \rightarrow X$  M(A, a)'dadır
    IF  $\lambda$  First(X)'te ise      THEN
      FOR Follow(A)'daki her eleman
         $A \rightarrow X$ 'i M(A, a)'ya ekle
```

# Örnek: LL(1) Ayırıştırma Tabloları Oluşturma

	First	Follow
exp	{(, num}	{\$, )}
exp'	{+, -, $\lambda$ }	{\$, )}
addop	{+, -}	{(, num}
term	{(, num}	{+, -, ), \$}
term'	{*, $\lambda$ }	{+, -, ), \$}
mulop	{*}	{(, num}
factor	{(, num}	{*, +, -, ), \$}

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

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

# LL(1) Gramer

---

- Bir gramer, LL(1) ayrıştırma tablosu her tablo girişinde en fazla bir kurala (production) sahipse, bir LL(1) gramerdir

# LL(1) olmayan Gramer için LL(1) Ayırıştırma Tablosu

1  $\text{exp} \rightarrow \text{exp addop term}$

2  $\text{exp} \rightarrow \text{term}$

3  $\text{term} \rightarrow \text{term mulop}$   
 $\text{factor}$

4  $\text{term} \rightarrow \text{factor}$

5  $\text{factor} \rightarrow ( \text{exp} )$

6  $\text{factor} \rightarrow \text{num}$

7  $\text{addop} \rightarrow +$

8  $\text{addop} \rightarrow -$

9  $\text{mulop} \rightarrow *$



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

$\text{First}(\text{exp}) = \{ (, \text{num} \}$

$\text{First}(\text{term}) = \{ (, \text{num} \}$

$\text{First}(\text{factor}) = \{ (, \text{num} \}$

$\text{First}(\text{addop}) = \{ +, - \}$

$\text{First}(\text{mulop}) = \{ * \}$

# LL(1) Olmayan Gramer Sorunları

---

- Grameri LL(1) yapmayan nelerdir?
  - Sol özyineleme (Left-recursion)
  - Sol faktör (Left factor)

# Left Recursion

---

- Yakın(Immediate) left recursion

- $A \rightarrow A X \mid Y$   $A=Y X^*$
- $A \rightarrow A X_1 \mid A X_2 \mid \dots \mid A X_n \mid Y_1 \mid Y_2 \mid \dots \mid Y_m$

$$A = \{Y_1, Y_2, \dots, Y_m\} \{X_1, X_2, \dots, X_n\}^*$$

- Yaygın(General) left recursion

- $A \Rightarrow X \Rightarrow^* A Y$

- Kolaylıkla kaldırılabilir

- $A \rightarrow Y A', A' \rightarrow X A' \mid \lambda$
- $A \rightarrow Y_1 A' \mid Y_2 A' \mid \dots \mid Y_m A', A' \rightarrow X_1 A' \mid X_2 A' \mid \dots \mid X_n A' \mid \lambda$

• Gramerde boş-string kuralı (production) ve çevrim (cycle) olmadığında kaldırılabilir (hiçbir programlama dili gramerinde görülmez)

# Yakın Left Recursion Kaldırma

---

$\text{exp} \rightarrow \text{exp} + \text{term} \mid \text{exp} - \text{term} \mid \text{term}$

$\text{term} \rightarrow \text{term} * \text{factor} \mid \text{factor}$

$\text{factor} \rightarrow ( \text{exp} ) \mid \text{num}$

• left recursion kaldır

$\text{exp} \rightarrow \text{term exp}'$        $\text{exp} = \text{term} (\pm \text{term})^*$

$\text{exp}' \rightarrow + \text{term exp}' \mid - \text{term exp}' \mid \lambda$

$\text{term} \rightarrow \text{factor term}'$

$\text{term}' \rightarrow * \text{factor term}' \mid \lambda$        $\text{term} = \text{factor} (* \text{factor})^*$

$\text{factor} \rightarrow ( \text{exp} ) \mid \text{num}$



# Left Factoring

---

- Left factor non-LL(1)'e sebep olur
  - $A \rightarrow X Y \mid X Z$  verilsin.  $A$  yığın tepesinde ve  $\text{First}(X)$ 'teki bir token sonraki token olduğunda hem  $A \rightarrow X Y$  hem de  $A \rightarrow X Z$  seçilebilir

$A \rightarrow X Y \mid X Z$

kuralı şu şekilde left-factor'lü olabilir

$A \rightarrow X A'$  ve  $A' \rightarrow Y \mid Z$

# Left Factor Örneği

---

$\text{ifSt} \rightarrow \text{if ( exp ) st else st} \mid \text{if ( exp ) st}$   
kuralı şu şekilde left-factor'lü olabilir

$\text{ifSt} \rightarrow \text{if ( exp ) st elsePart}$   
 $\text{elsePart} \rightarrow \text{else st} \mid \lambda$

$\text{seq} \rightarrow \text{st ; seq} \mid \text{st}$   
kuralı şu şekilde left-factor'lü olabilir

$\text{seq} \rightarrow \text{st seq'}$   
 $\text{seq'} \rightarrow \text{ ; seq} \mid \lambda$

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing)

---

- Ayırıştırma problemi bir sağ–cümlesel formda türevde önceki sağ–cümlesel formu elde etmek için azaltılacak doğru sağ kısmı (RHS) bulmaktır
- Ayırıştırma için açık yığın kullanır
- Sağdan (rightmost) türetmeyi (**R**) soldan (left) (**L**) sağa simüle eder, böylece LR ayırıştırma olarak bilinir
- Yukarıdan–Aşağıya ayırtırmadan daha güçlüdür
  - Sol özyineleme problem olmaz
- İki hareket
  - Shift: sonraki giriş token'ını yığına al
  - Reduce:  $A \rightarrow B$  kuralı verildiğinde yığının üstündeki bir B stringini terminal olmayan bir A ile yer değiş

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing)

---

int \* int + int

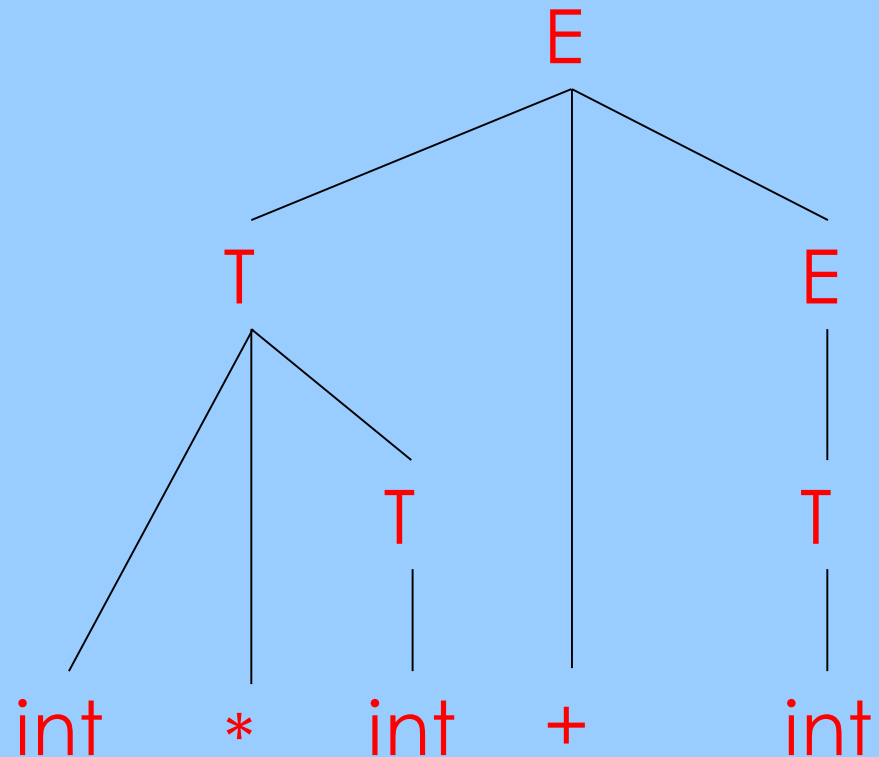
int \* T + int

T + int

T + T

T + E

E



# Örnek 1: Aşağıdan–Yukarıya Ayrıştırma (1)

---

int \* int + int

int \* int + int

# Aşağıdan-Yukarıya Ayırıştırma (2)

---

int \* int + int

int \* T + int

int \* int + int

T

|

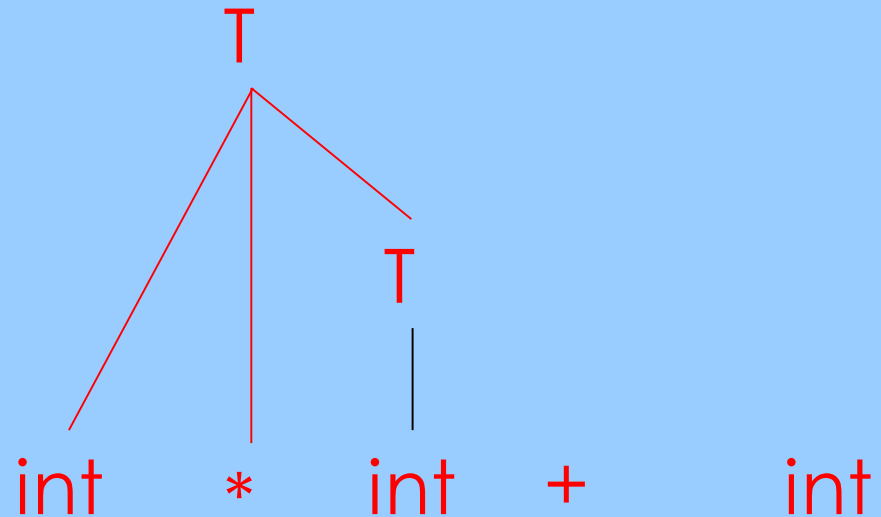
# Aşağıdan-Yukarıya Ayırıştırma (3)

---

int \* int + int

int \* T + int

T + int



# Aşağıdan-Yukarıya Ayırıştırma (4)

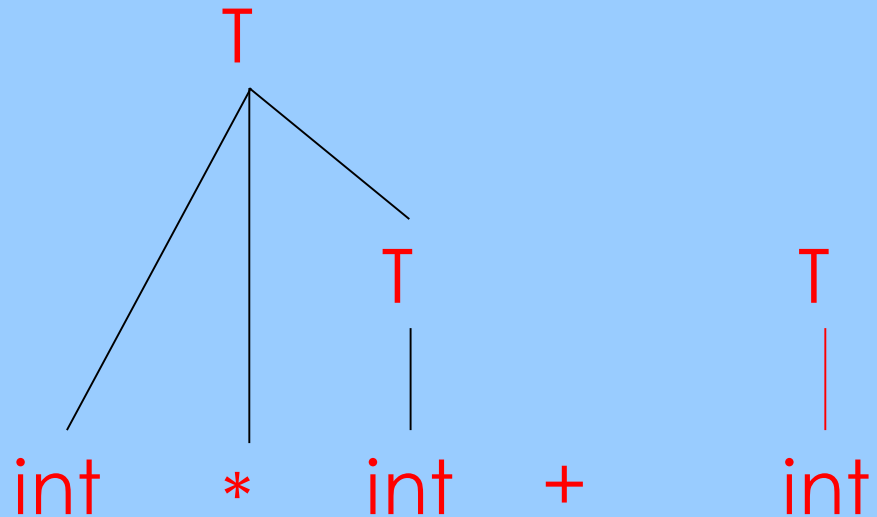
---

int \* int + int

int \* T + int

T + int

T + T





# Aşağıdan-Yukarıya Ayırıştırma (5)

---

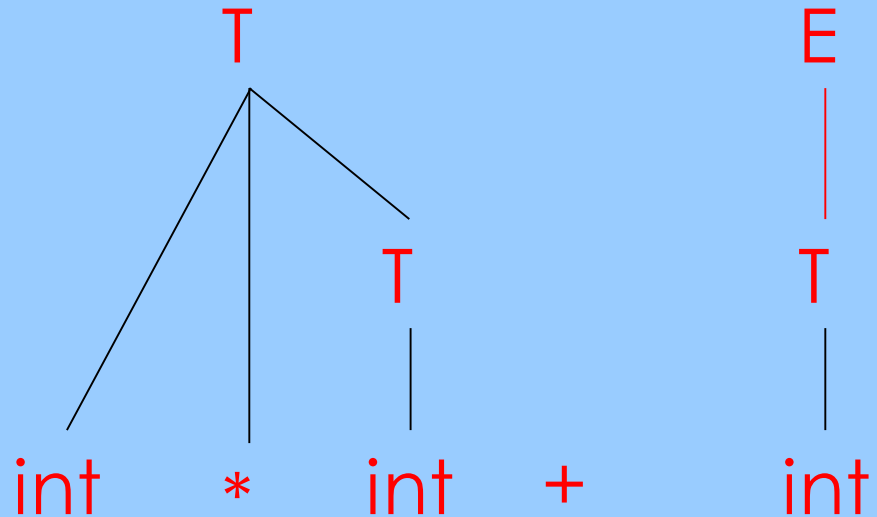
int \* int + int

int \* T + int

T + int

T + T

T + E



# Aşağıdan-Yukarıya Ayırıştırma (6)

---

int \* int + int

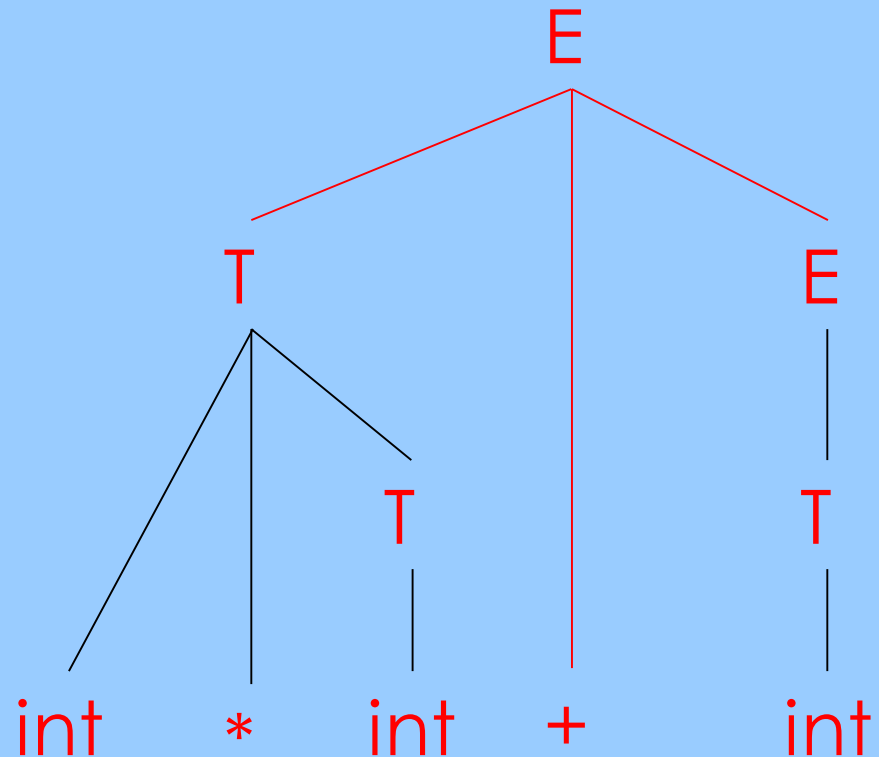
int \* T + int

T + int

T + T

T + E

E



## Örnek 2: Aşağıdan–Yukarıya Ayrıştırma (1)

---

int + (int) + (int)

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$

int + ( int ) + ( int )

# Aşağıdan-Yukarıya Ayırıştırma (2)

**int** + (int) + (int)

E + (int) + (int)

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$

E  
|  
int + ( int ) + ( int )

# Aşağıdan-Yukarıya Ayırıştırma (3)

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$

$\text{int} + (\text{int}) + (\text{int})$

$E + (\text{int}) + (\text{int})$

$E + (E) + (\text{int})$

$$\begin{array}{ccccccc} E & & E & & & & \\ | & & | & & & & \\ \text{int} & + & ( & \text{int} & ) & + & ( & \text{int} & ) \end{array}$$

# Aşağıdan-Yukarıya Ayırıştırma (4)

$E \rightarrow \text{int}$

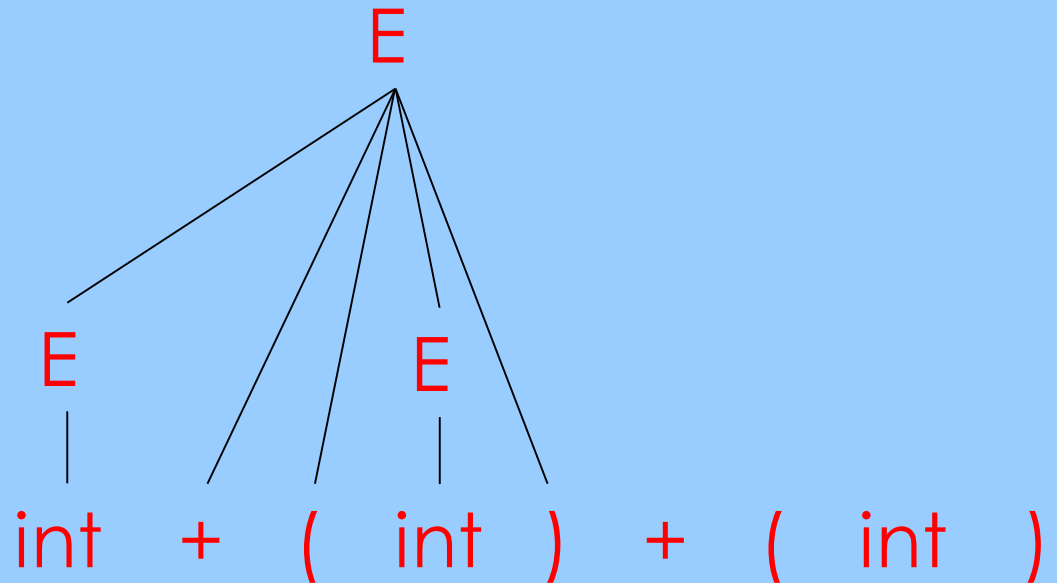
$E \rightarrow E + (E)$

$\text{int} + (\text{int}) + (\text{int})$

$E + (\text{int}) + (\text{int})$

$E + (E) + (\text{int})$

$E + (\text{int})$



# Aşağıdan-Yukarıya Ayırıştırma (5)

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$

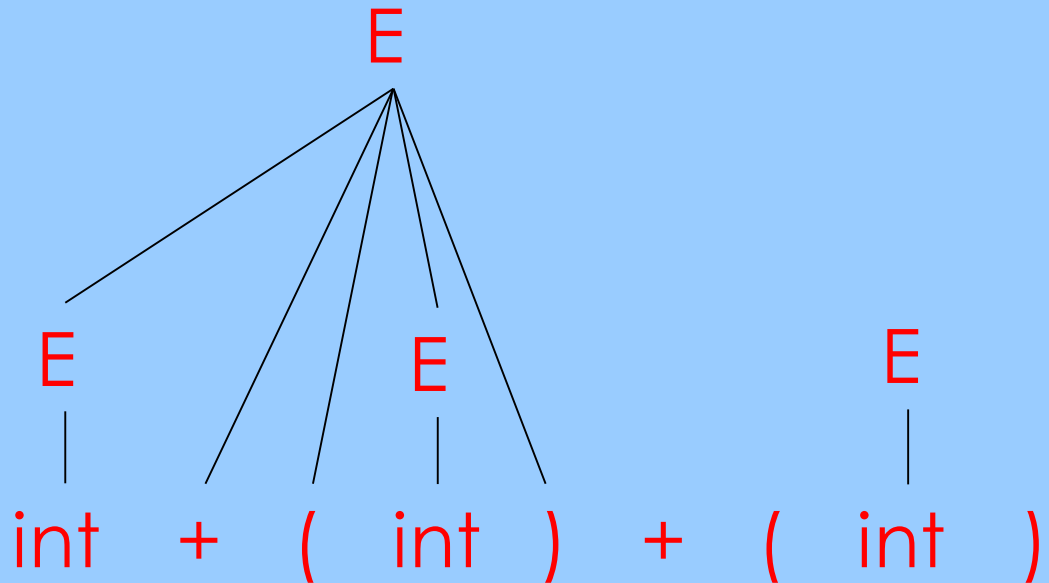
$\text{int} + (\text{int}) + (\text{int})$

$E + (\text{int}) + (\text{int})$

$E + (E) + (\text{int})$

$E + (\text{int})$

$E + (E)$



# Aşağıdan-Yukarıya Ayırıştırma (6)

int + (int) + (int)

E + (int) + (int)

E + (E) + (int)

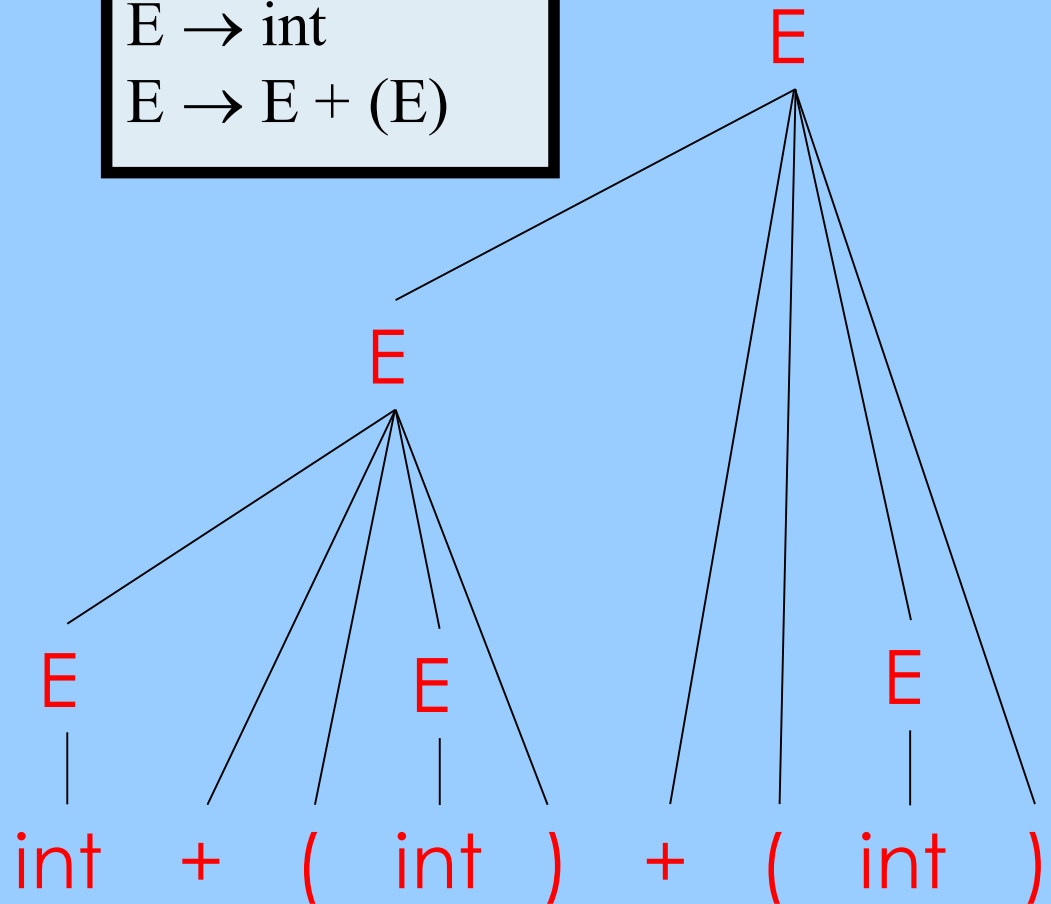
E + (int)

E + (E)

E

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$





# Basit bir Aşağıdan–Yukarıya Ayırıştırma Algoritması

---

$I$  = giriş stringi olsun

repeat

$I$ 'nın boş olmayan bir  $\beta$  altstringini

$X \rightarrow \beta$  kuralından al

böyle bir  $\beta$  yoksa, geldiğin yoldan geri  
dön (backtrack)

$I$ 'da  $X$  kullanarak bir  $\beta$  yerleştir

until  $I = "S"$  (başlangıç sembolü) ya da tüm  
olasılıklar tükenir

# Sorular

---

- Algoritma sonlanır mı?
- Algoritma ne kadar hızlıdır?
- Algoritma tüm durumları ele alır mı?
- Her adımda reduce yapmak için altstringleri nasıl seçeriz?

# Notasyon

---

- Fikir: Stringi iki alt stringe ayır
  - Sağ altsitringe henüz ayrıştırma işlemi tetkik edilmemiş (terminaller stringi)
  - Sol altstrin terminal ve terminal olmayan öğeler sahip
- Bölme noktası | ile gösterilsin
  - | stringin parçacı değildir
- Başlangıçta, giriş tetkik edilmemiş  $x_1 x_2 \dots x_n$

# Shift-Reduce Parsing

---

Aşağıdan Yukarıya ayrıştırma sadece iki tip hareket kullanır: *Shift* ve *Reduce*

- Shift: |'yı bir sağa hareket ettir

- Bir terminali sol altstringe kaydır

$$ABC|xyz \Rightarrow ABCx|yz$$

$$E + ( | \text{int} ) \Rightarrow E + ( \text{int} | )$$

- Reduce: Sol altstringin sağında *ters kural* (üretim)

- $A \rightarrow xy$  bir kurala, o zaman

$$Cbxy|ijk \Rightarrow CbA|ijk$$

- $E \rightarrow E + ( E )$  bir kurala, o zaman

$$E + ( \underline{E + ( E )} | ) \Rightarrow E + ( \underline{E} | )$$

# Sadece Reduce Örneği

---

int \* int | + int

int \* T | + int

reduce  $T \rightarrow \text{int}$

reduce  $T \rightarrow \text{int} * T$

T + int |

T + T |

T + E |

E |

reduce  $T \rightarrow \text{int}$

reduce  $E \rightarrow T$

reduce  $E \rightarrow T + E$

# Shift-Reduce Ayırıştırma Örneği

---

int * int + int	shift
int   * int + int	shift
int *   int + int	shift
int * int   + int	reduce $T \rightarrow \text{int}$
int * T   + int	reduce $T \rightarrow \text{int} * T$
T   + int	shift
T +   int	shift
T + int	reduce $T \rightarrow \text{int}$
T + T	reduce $E \rightarrow T$
T + E	reduce $E \rightarrow T + E$
E	

# Örnek 1: Shift-Reduce Ayırıştırma (1)

---

| int \* int + int

↑ int \* int + int

# Örnek: Shift-Reduce Ayırıştırma (2)

---

| int \* int + int

int | \* int + int

int \* int + int  
↑



# Örnek: Shift-Reduce Ayırıştırma (3)

---

| int \* int + int

int | \* int + int

int \* | int + int

int \* int + int  
          ↑

# Örnek: Shift-Reduce Ayırıştırma (4)

---

| int \* int + int

int | \* int + int

int \* | int + int

int \* int | + int

int \* int + int

↑

# Örnek: Shift-Reduce Ayırıştırma (5)

---

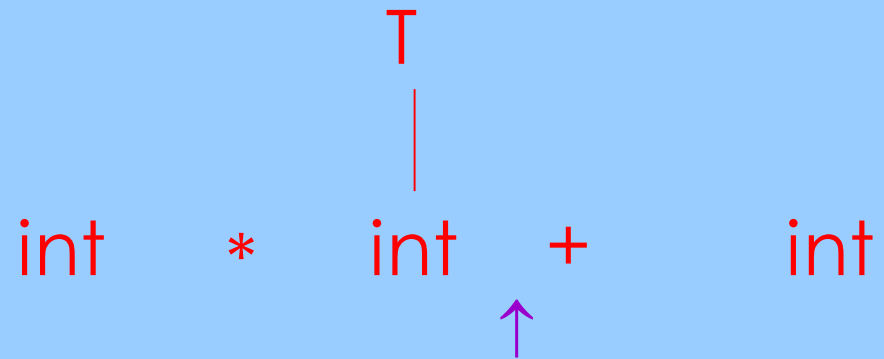
| int \* int + int

int | \* int + int

int \* | int + int

int \* int | + int

int \* T | + int



# Örnek: Shift-Reduce Ayırıştırma (6)

---

| int \* int + int

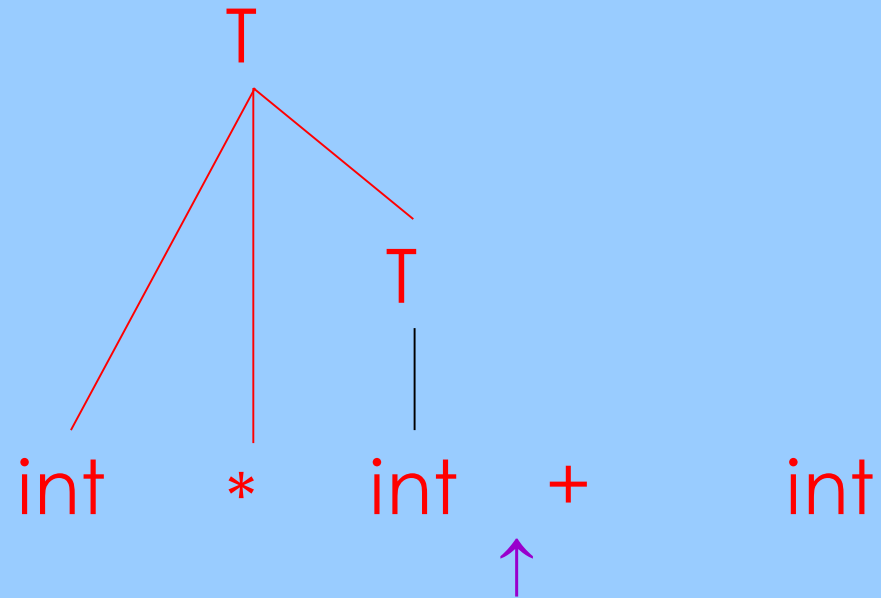
int | \* int + int

int \* | int + int

int \* int | + int

int \* T | + int

T | + int



# Örnek: Shift-Reduce Ayırıştırma (7)

---

| int \* int + int

int | \* int + int

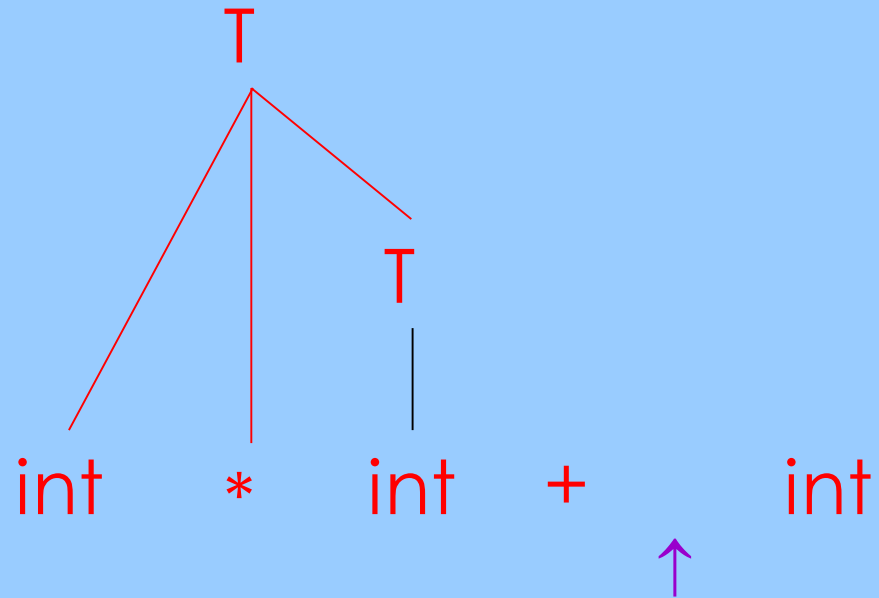
int \* | int + int

int \* int | + int

int \* T | + int

T | + int

T + | int



# Örnek: Shift-Reduce Ayırıştırma (8)

---

| int \* int + int

int | \* int + int

int \* | int + int

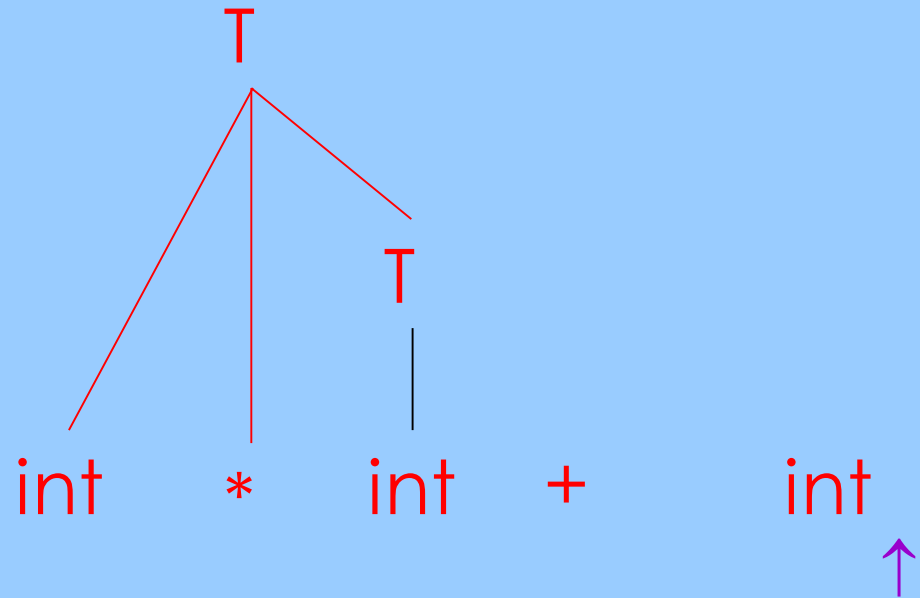
int \* int | + int

int \* T | + int

T | + int

T + | int

T + int |



# Örnek: Shift-Reduce Ayırıştırma (9)

---

| int \* int + int

int | \* int + int

int \* | int + int

int \* int | + int

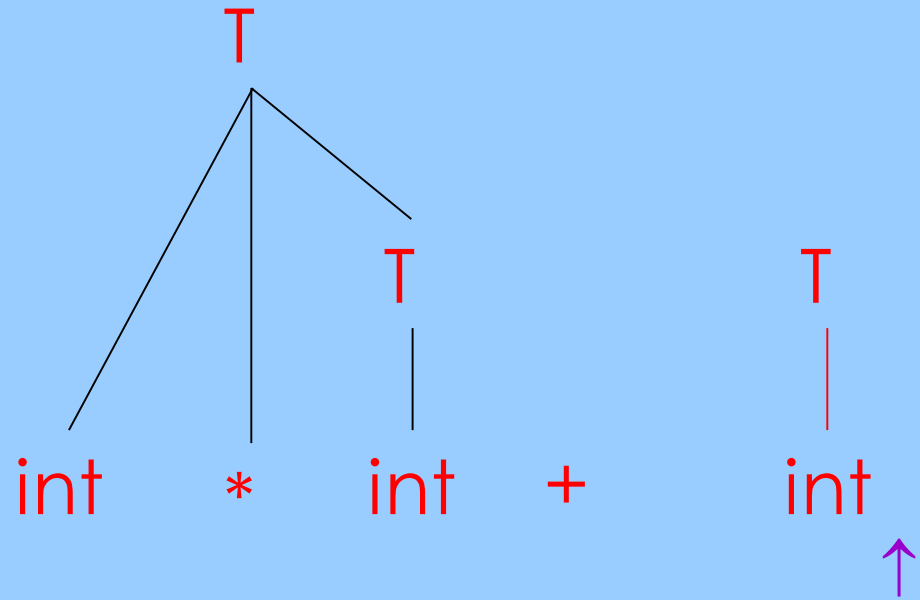
int \* T | + int

T | + int

T + | int

T + int |

T + T |



# Örnek: Shift-Reduce Ayırıştırma (10)

| int \* int + int

int | \* int + int

int \* | int + int

int \* int | + int

int \* T | + int

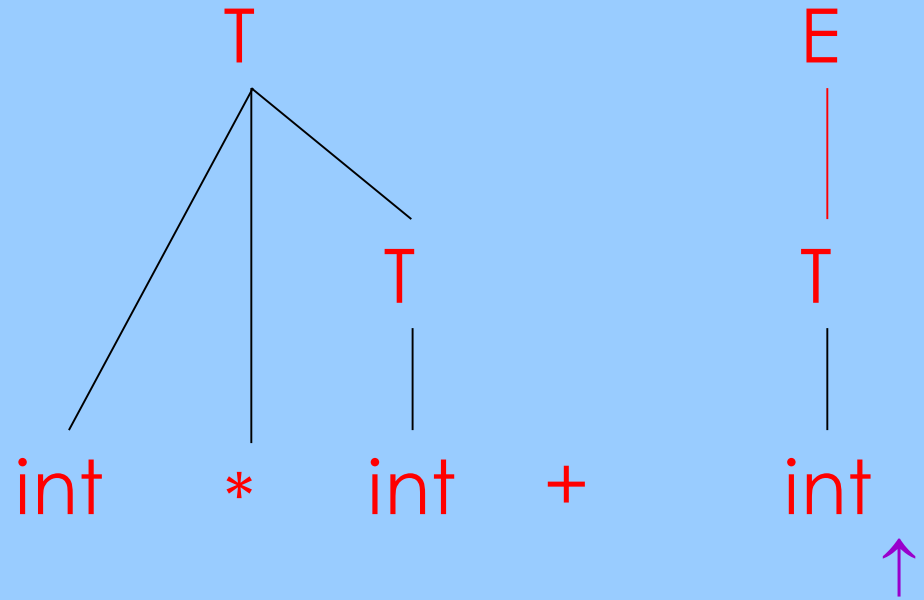
T | + int

T + | int

T + int |

T + T |

T + E |





# Örnek: Shift-Reduce Ayırıştırma (1 1)

| int \* int + int

int | \* int + int

int \* | int + int

int \* int | + int

int \* T | + int

T | + int

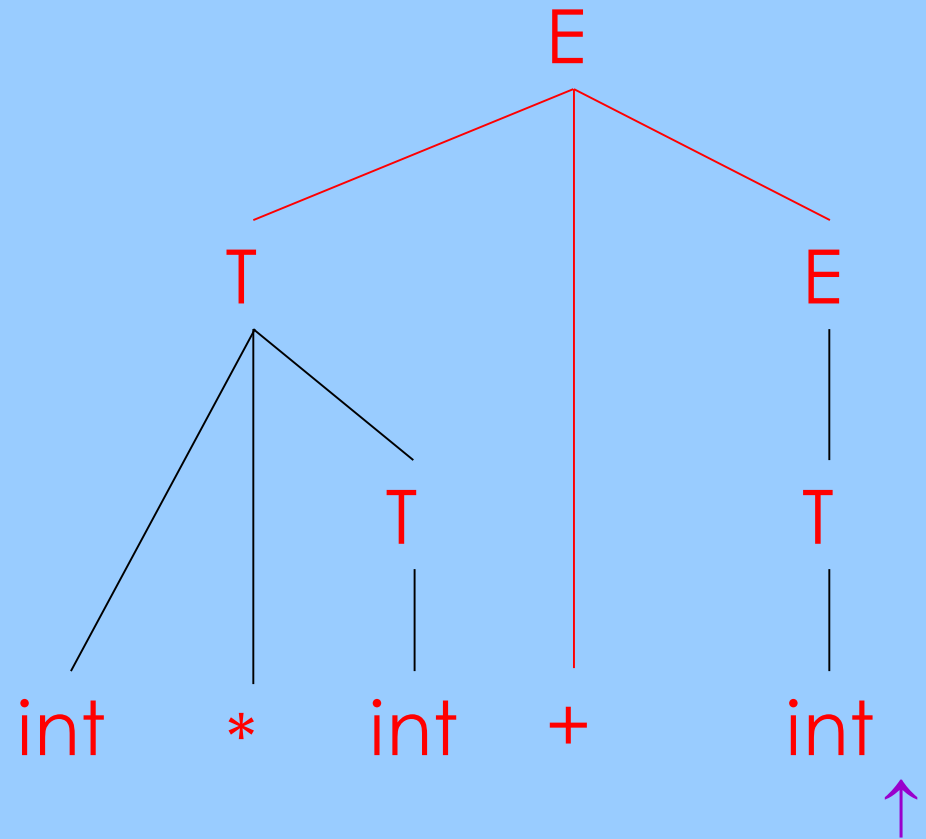
T + | int

T + int |

T + T |

T + E |

E |



# Örnek 2: Shift-Reduce Ayırıştırma (1)

↑ int + (int) + (int)\$ shift

$E \rightarrow \text{int}$

$E \rightarrow E + (E)$

int + ( int ) + ( int )



# Örnek: Shift-Reduce Ayırıştırma (2)

↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow int$

$E \rightarrow int$   
 $E \rightarrow E + (E)$

int + ( int ) + ( int )  
↑

# Örnek: Shift-Reduce Ayırıştırma (3)

↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E ↑ + (int) + (int)\$ shift 3 kez

$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$

E  
/  
int + ( int ) + ( int )  
↑

# Örnek: Shift-Reduce Ayırıştırma (4)

↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E ↑ + (int) + (int)\$ shift 3 kez  
E + (int ↑ ) + (int)\$ red.  $E \rightarrow \text{int}$

$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$

E  
/  
int + ( int ) + ( int )  
↑

# Örnek: Shift-Reduce Ayırıştırma (5)

$\uparrow \text{int} + (\text{int}) + (\text{int})\$$  shift  
 $\text{int} \uparrow + (\text{int}) + (\text{int})\$$  red.  $E \rightarrow \text{int}$   
 $E \uparrow + (\text{int}) + (\text{int})\$$  shift 3 kez  
 $E + (\text{int} \uparrow) + (\text{int})\$$  red.  $E \rightarrow \text{int}$   
 $E + (E \uparrow) + (\text{int})\$$  shift

$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$

$$\begin{array}{ccccccc} & E & & E & & & \\ & / & & / & & & \\ \text{int} & + & ( & \text{int} & ) & + & ( & \text{int} & ) \end{array}$$

↑

# Örnek: Shift-Reduce Ayırıştırma (6)

$\uparrow \text{int} + (\text{int}) + (\text{int})\$$  shift  
 $\text{int} \uparrow + (\text{int}) + (\text{int})\$$  red.  $E \rightarrow \text{int}$   
 $E \uparrow + (\text{int}) + (\text{int})\$$  shift 3 kez  
 $E + (\text{int} \uparrow) + (\text{int})\$$  red.  $E \rightarrow \text{int}$   
 $E + (E \uparrow) + (\text{int})\$$  shift  
 $E + (E) \uparrow + (\text{int})\$$  red.  $E \rightarrow E + (E)$

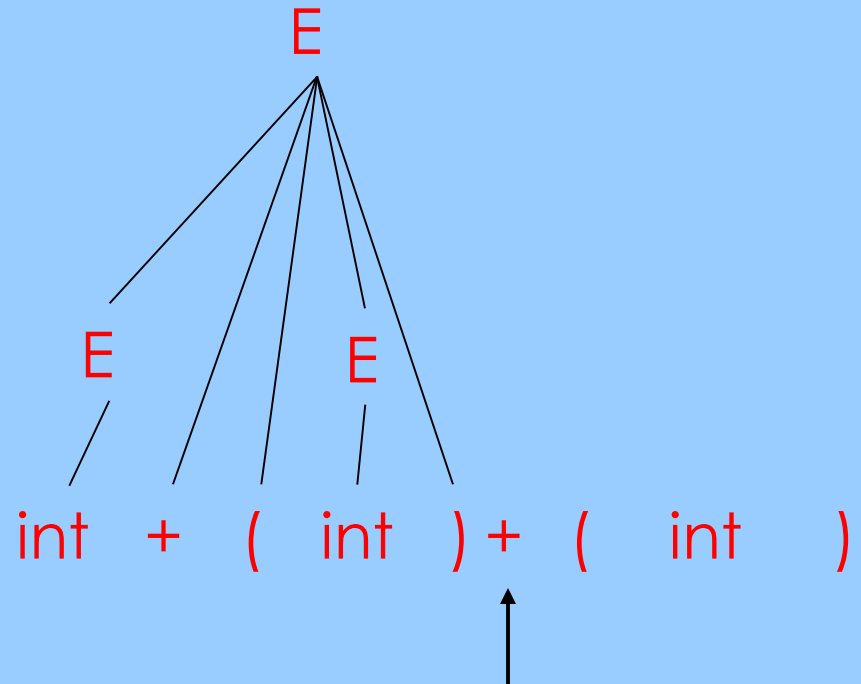
$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$

$$\begin{array}{ccccccc} & E & & E & & & \\ & / & & / & & & \\ \text{int} & + & ( & \text{int} & ) & + & ( & \text{int} & ) \\ & & & & & & \uparrow & & \end{array}$$

# Örnek: Shift-Reduce Ayırıştırma (7)

↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E ↑ + (int) + (int)\$ shift 3 kez  
E + (int ↑) + (int)\$ red.  $E \rightarrow \text{int}$   
E + (E ↑) + (int)\$ shift  
E + (E) ↑ + (int)\$ red.  $E \rightarrow E + (E)$   
E ↑ + (int)\$ shift 3 kez

$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$

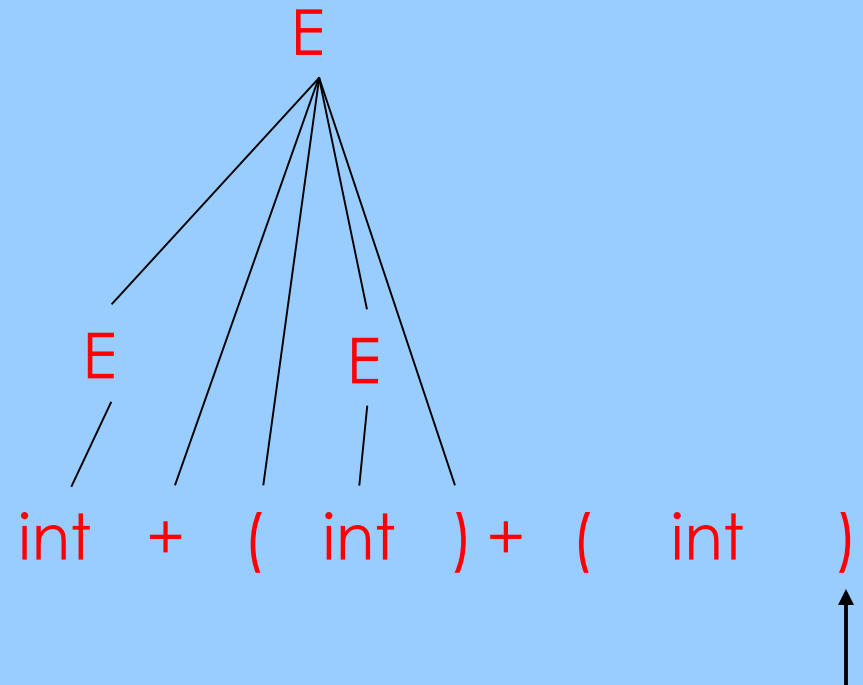




# Örnek: Shift-Reduce Ayırıştırma (8)

$\uparrow$  int + (int) + (int)\$ shift  
int  $\uparrow$  + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E  $\uparrow$  + (int) + (int)\$ shift 3 kez  
E + (int  $\uparrow$  ) + (int)\$ red.  $E \rightarrow \text{int}$   
E + (E  $\uparrow$  ) + (int)\$ shift  
E + (E)  $\uparrow$  + (int)\$ red.  $E \rightarrow E + (E)$   
E  $\uparrow$  + (int)\$ shift 3 kez  
E + (int  $\uparrow$  )\$ red.  $E \rightarrow \text{int}$

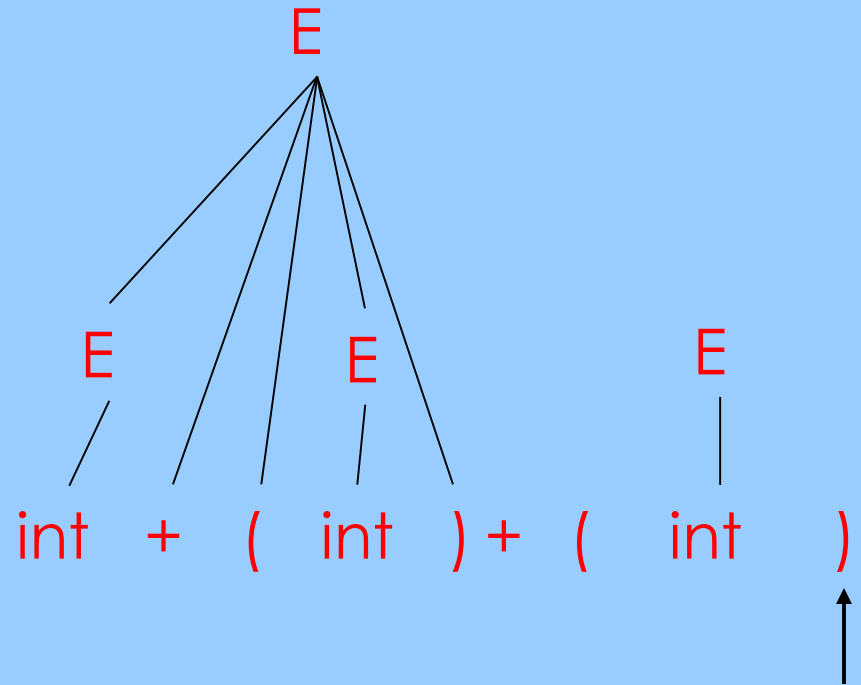
$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$



# Örnek: Shift-Reduce Ayırıştırma (9)

↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E ↑ + (int) + (int)\$ shift 3 kez  
E + (int ↑) + (int)\$ red.  $E \rightarrow \text{int}$   
E + (E ↑) + (int)\$ shift  
E + (E) ↑ + (int)\$ red.  $E \rightarrow E + (E)$   
E ↑ + (int)\$ shift 3 kez  
E + (int ↑)\$ red.  $E \rightarrow \text{int}$   
E + (E ↑)\$ shift

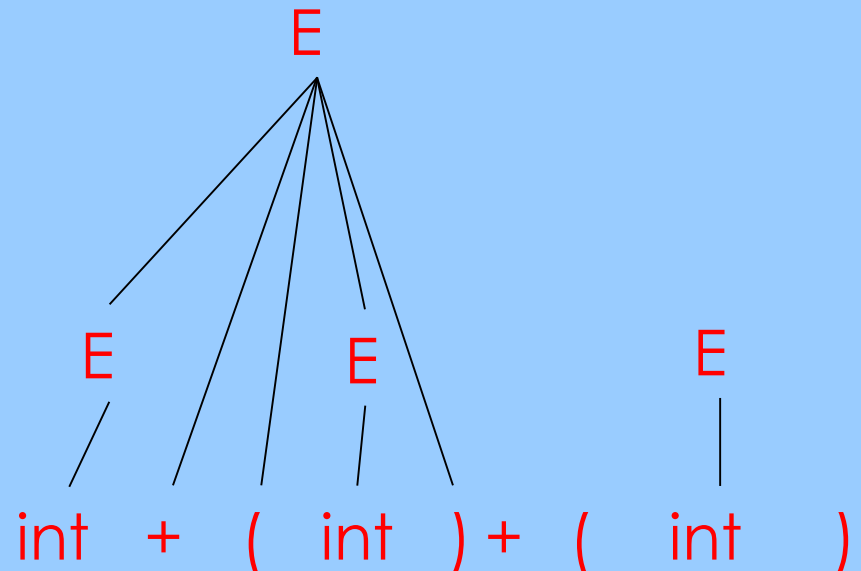
$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$



# Örnek: Shift-Reduce Ayırıştırma (10)

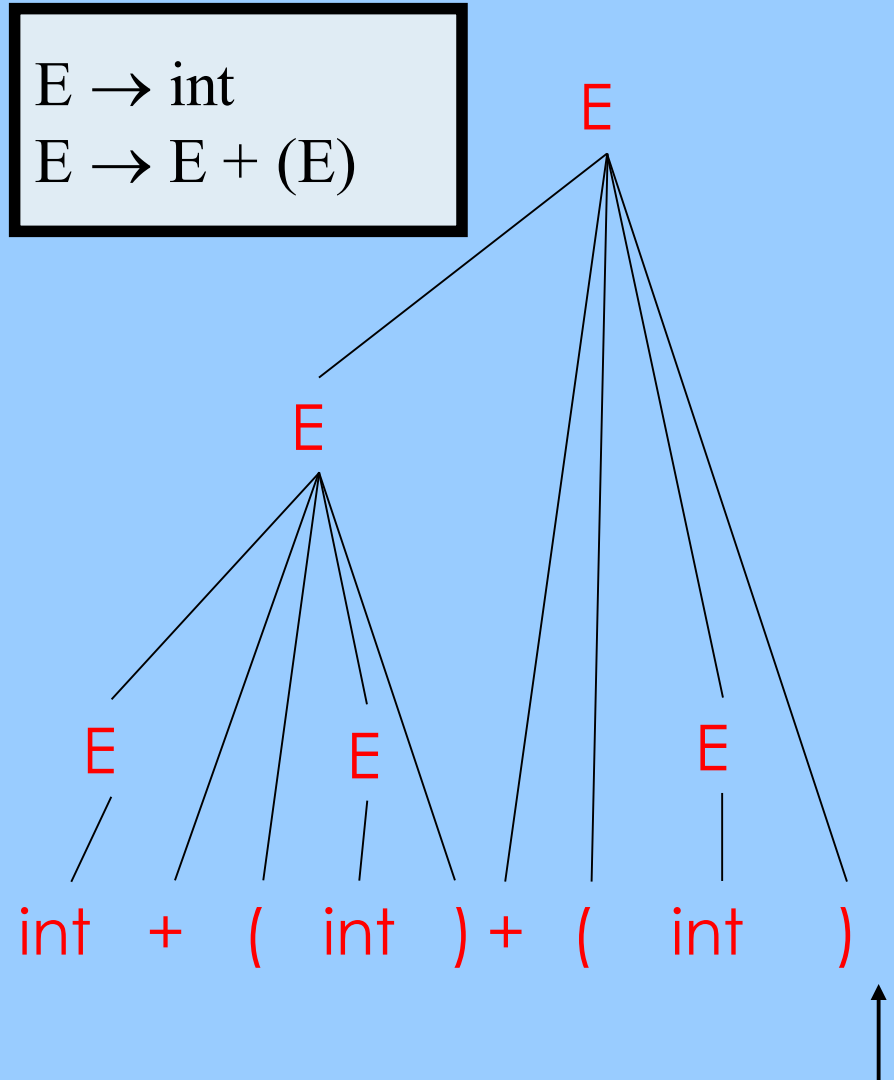
↑ int + (int) + (int)\$ shift  
int ↑ + (int) + (int)\$ red.  $E \rightarrow \text{int}$   
E ↑ + (int) + (int)\$ shift 3 kez  
E + (int ↑) + (int)\$ red.  $E \rightarrow \text{int}$   
E + (E ↑) + (int)\$ shift  
E + (E) ↑ + (int)\$ red.  $E \rightarrow E + (E)$   
E ↑ + (int)\$ shift 3 kez  
E + (int ↑)\$ red.  $E \rightarrow \text{int}$   
E + (E ↑)\$ shift  
E + (E) ↑ \$ red.  $E \rightarrow E + (E)$

$E \rightarrow \text{int}$   
 $E \rightarrow E + (E)$



# Örnek: Shift-Reduce Ayırıştırma (1 1)

$\uparrow \text{int} + (\text{int}) + (\text{int})\$$	shift
$\text{int} \uparrow + (\text{int}) + (\text{int})\$$	red. $E \rightarrow \text{int}$
$E \uparrow + (\text{int}) + (\text{int})\$$	shift 3 kez
$E + (\text{int} \uparrow) + (\text{int})\$$	red. $E \rightarrow \text{int}$
$E + (E \uparrow) + (\text{int})\$$	shift
$E + (E) \uparrow + (\text{int})\$$	red. $E \rightarrow E + (E)$
$E \uparrow + (\text{int})\$$	shift 3 kez
$E + (\text{int} \uparrow) \$$	red. $E \rightarrow \text{int}$
$E + (E \uparrow) \$$	shift
$E + (E) \uparrow \$$	red. $E \rightarrow E + (E)$
$E \uparrow \$$	accept



# Örnek 3: Shift-Reduce Ayırıştırma

## Örneği

Stack	Input	Action
\$	a b b c d e \$	Shift
\$ a	b b c d e \$	Shift
\$ a b	b c d e \$	Reduce $A \Rightarrow b$
\$ a A	b c d e \$	Shift
\$ a A b	c d e \$	Shift
\$ a A b c	d e \$	Reduce $A \Rightarrow A b c$
\$ a A	d e \$	Shift
\$ a A d	e \$	Reduce $B \Rightarrow d$
\$ a A B	e \$	Shift
\$ a A B e	\$	Reduce $S \Rightarrow a A B$
\$	\$	e

$S \Rightarrow a A B e$

$A \Rightarrow A b c \mid b$

$B \Rightarrow d$

# İzi nasıl koruruz?

---

Sol parça stringi **yığın (stack)** olarak uygulanır

- Yığının tepesinde ↑ vardır
- **Shift:**
  - Yığına bir terminal koyar
- **Reduce:**
  - Yığından 0 ya da daha fazla sembol alır
  - Bu semboller bir kuralın sağ tarafındadır
  - Yığına bir terminal olmayan koyar (production LHS)

# LR

parse "a b b c d e"

Grammer:

$S \rightarrow a A B e$

$A \rightarrow A b c \mid b$

$B \rightarrow d$

Bir cümle indirgeme:

a b b c d e

a A b c d e

a A d e

a A B e

S

Ağaç bir sağ türetmeye uyar:

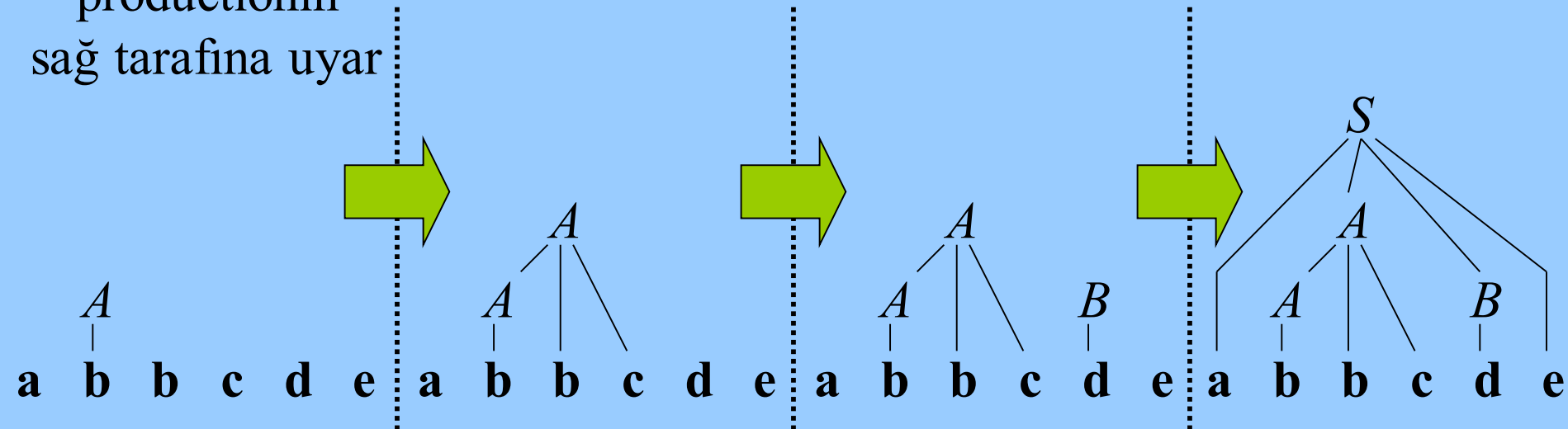
$S \Rightarrow a A B e$

$\Rightarrow a A d e$

$\Rightarrow a A b c d e$

$\Rightarrow a b b c d e$

Bunlar  
productionın  
sağ tarafına uyar



# LR(0) Ayırıştırıcı

---

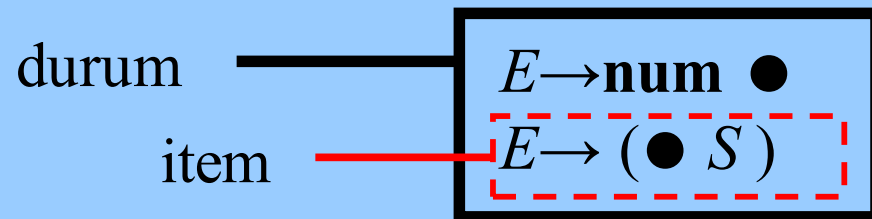
- Soldan-sağa (Left-to-right) tarama, sağ (Right-most) türetme, “sıfır” ileri bakma (look-ahead) karakteri
- Çoğu dil gramerlerini idarede çok zayıf (*mesela*, “sum” grameri)
- Fakat fazla karışık ayırıştırıcılar için bize ön hazırlık yapmaya yardım eder



# LR(0) Durumları

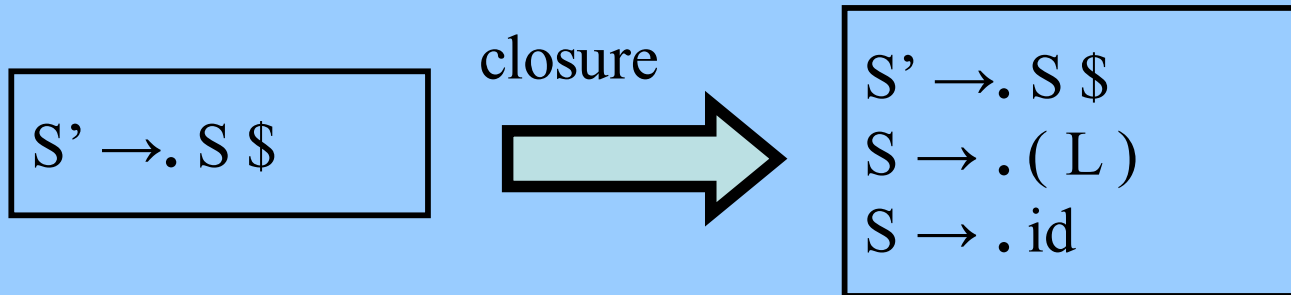
---

- Bir durum olası indirgemeler için ilerleme izini koruyan *item*lar kümesidir
- Bir  $LR(0)$  item'i i kuralın sağ tarafında bir yerde "." ayıracıyla ayrılmış dilden bir kuraldır s a production



- "."dan önceki şeyler: zaten yığındadır (indirgenecek olası  $\gamma$ 'lerin başlangıcı)
- "."dan sonraki şeyler: bir sonra bakacağımızdır
- $\alpha$  öneki (prefixi) durumun kendisiyle temsil edilir

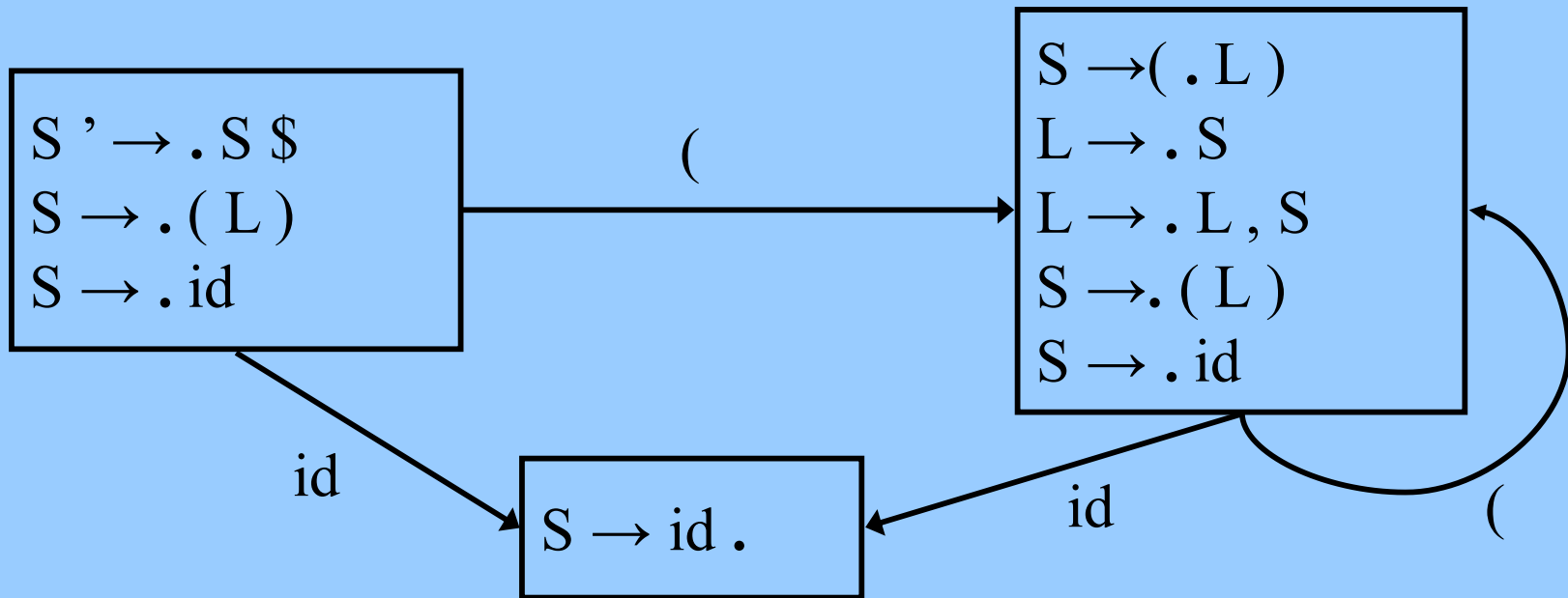
# Başlangıç Durumu & Kapalılık (Closure)

$$\begin{array}{l} S \rightarrow ( L ) \mid id \\ L \rightarrow S \mid L, S \end{array}$$


Bir yığını okuyacak DFA inşası:

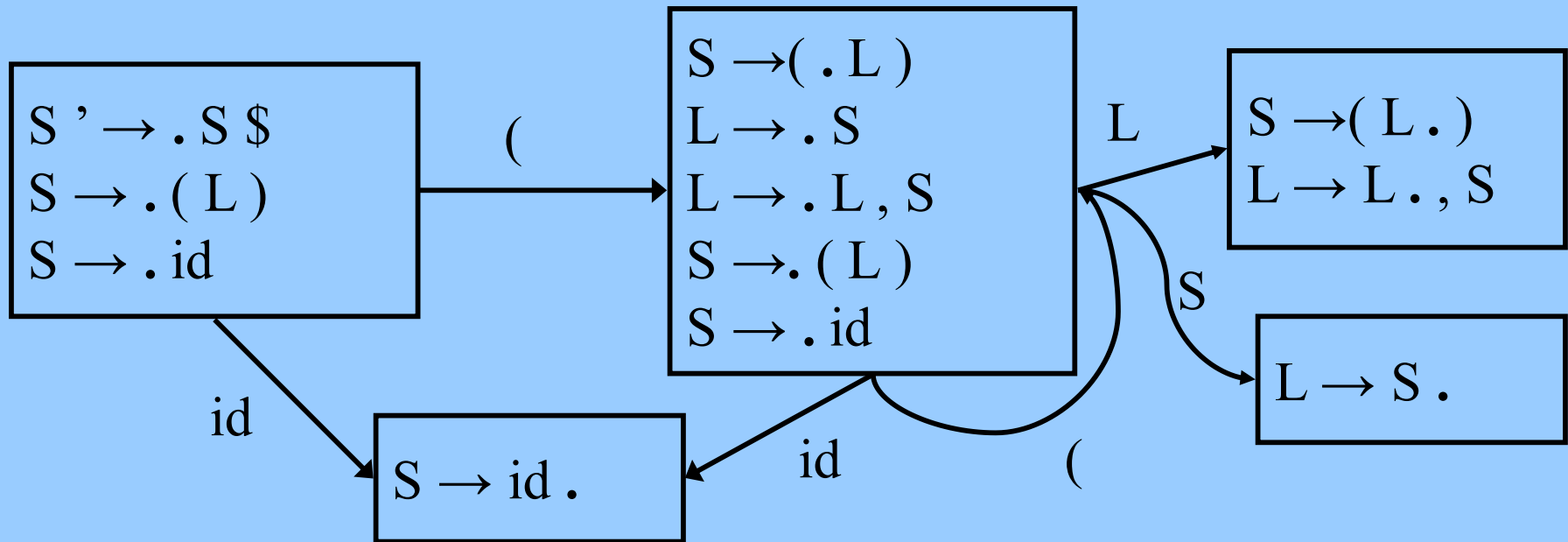
- İlk adım:  $S' \rightarrow S \$$  kurallı augment gramer
- DFA'nın başlangıç durumu: boş yığın =  $S' \rightarrow \cdot S \$$
- Bir durumu *Closure'u* sol tarafı durumda bir item'da bulunan tüm kurallar için "."dan hemen sonra item'lar ekler
  - Bir sonra indirgenecek olası kurallar kümesi
  - Eklenen itemlar başta yerleşmiş "."ya sahiptir: yığındaki bu itemlar için henüz sembol yok

# Terminal Semboller Uygulama

$$S \rightarrow ( L ) \mid id$$
$$L \rightarrow S \mid L , S$$


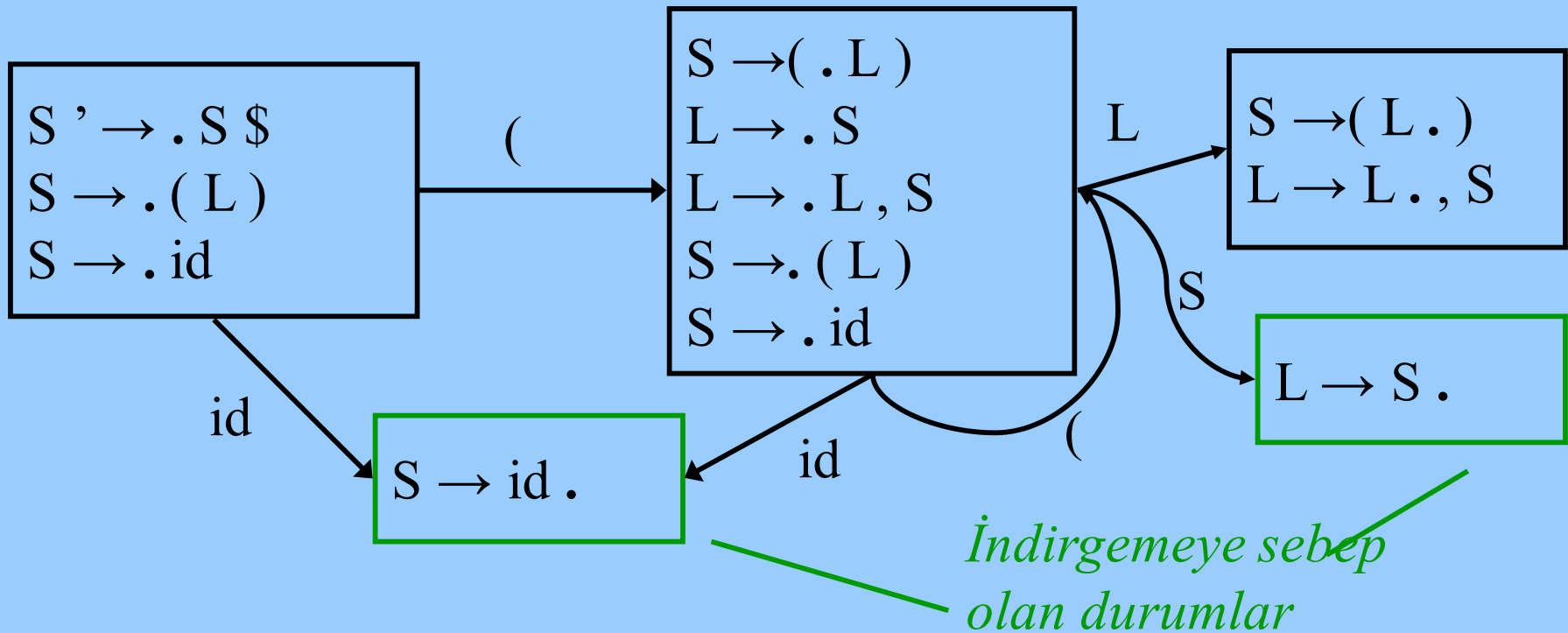
Yeni durumda, noktadan hemen sonra uygun giriş sembolüne sahip tüm itemları dahil et, bu itemlar üzerinde ilerle ve, *closure*'u kabul et.

# Terminal Olmayan Semboller Uygulama

$$S \rightarrow (L) \mid id$$
$$L \rightarrow S \mid L, S$$


- Yığındaki terminal olmayan sembolere sanki terminalmiş gibi davranılır (indirgeme ile eklenenler hariç)

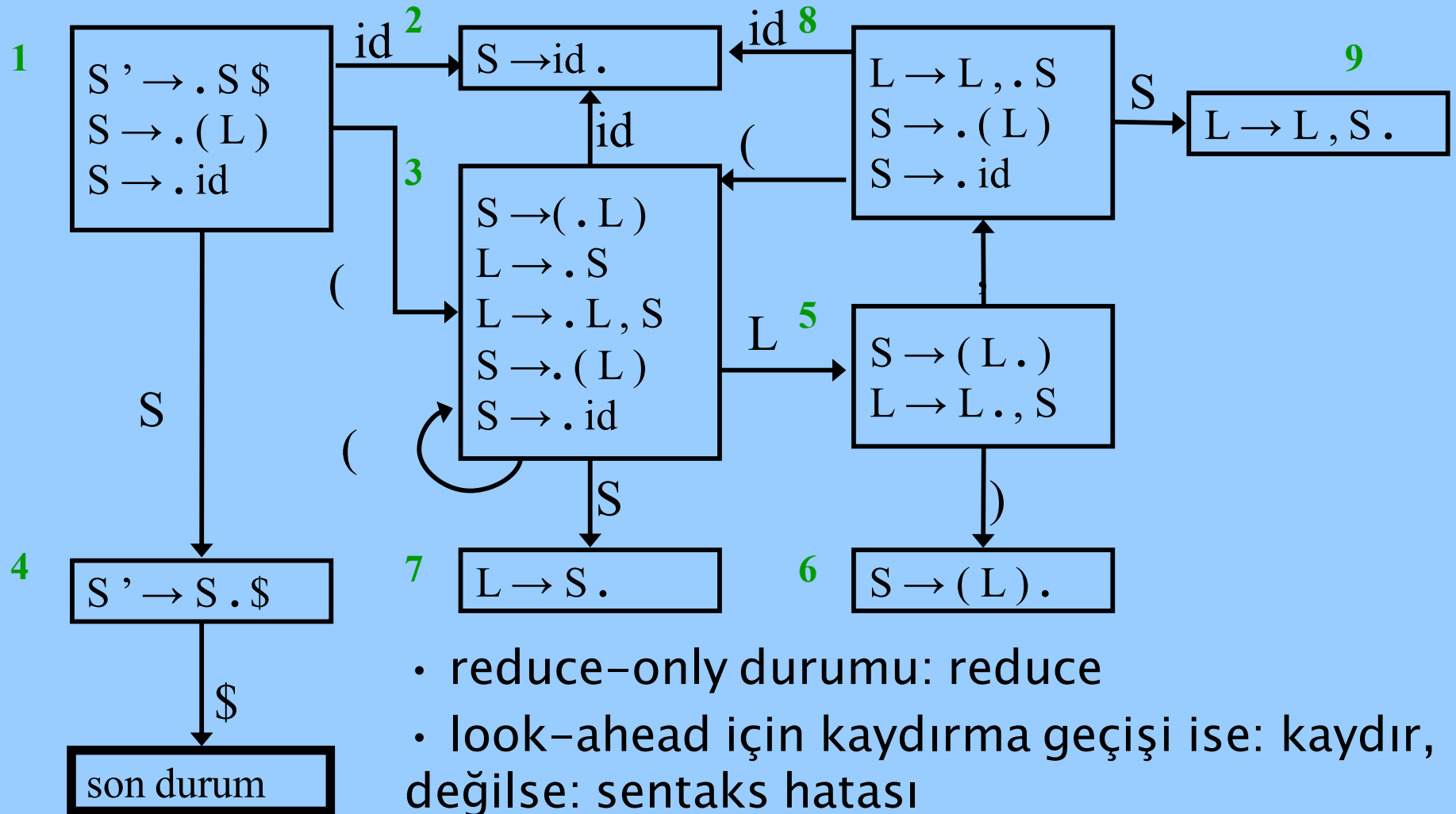
# Reduce Hareketini Uygulama

$$S \rightarrow ( L ) \mid id$$
$$L \rightarrow S \mid L, S$$


- Yığından sağ tarafı al, sol tarafla yer değiştir  
 $X (X \rightarrow Y)$

# Tam DFA

$S \rightarrow (L) \mid id$   
 $L \rightarrow S \mid L, S$



# Ayrıştırma Örneği: ((x),y)

$$S \rightarrow (L) \mid id$$

$$L \rightarrow S \mid L, S$$

*türetme*

((x),y) ←

((x),y) ←

((x),y) ←

((x),y) ←

((S),y) ←

((L),y) ←

((L),y) ←

(S,y) ←

(L,y) ←

(L,y) ←

(L,y) ←

(L,S) ←

(L) ←

(L) ←

S

*yığın*

1

1 (3

1 (3 (3

1 (3 (3 x2

1 (3 (3 S7

1 (3 (3 L5

1 (3 (3 L5)6

1 (3 S7

1 (3 L5

1 (3 L5 , 8

1 (3 L5 , 8 y2

1 (3 L5 , 8 S9

1 (3 L5

1 (3 L5 )6

1 S4

*giriş*

((x),y)

(x),y)

x),y)

),y)

),y)

),y)

,y)

,y)

,y)

y)

)

)

)

\$

*hareket*

shift, goto 3

shift, goto 3

shift, goto 2

reduce S→id

reduce L→S

shift, goto 6

reduce S→(L)

reduce L→S

shift, goto 8

shift, goto 9

reduce S→id

reduce L→L,S

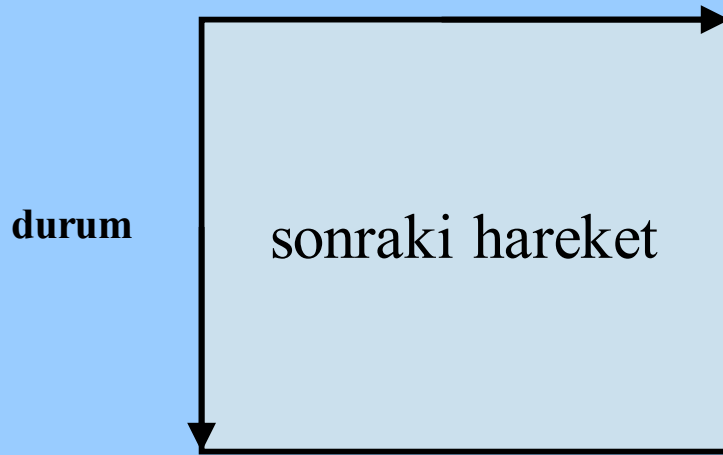
shift, goto 6

reduce S→(L)

done

# Gerçekleştirme: LR Ayırıştırma Tablosu

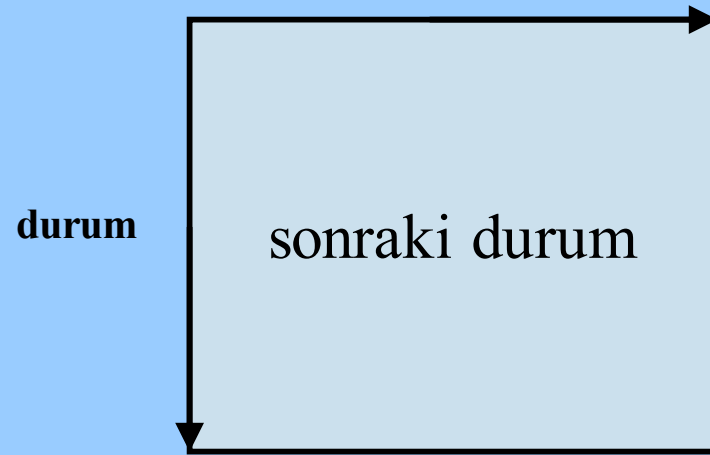
giriş (terminal) sembolleri



**Hareket (Action) tablosu**

Shift ya da reduce'a karar verecek her adımda kullanılır

terminal olmayan semboller



**Git (Goto) tablosu**

Bir sonraki duruma karar vermek için sadece reduce'ta kullanılır

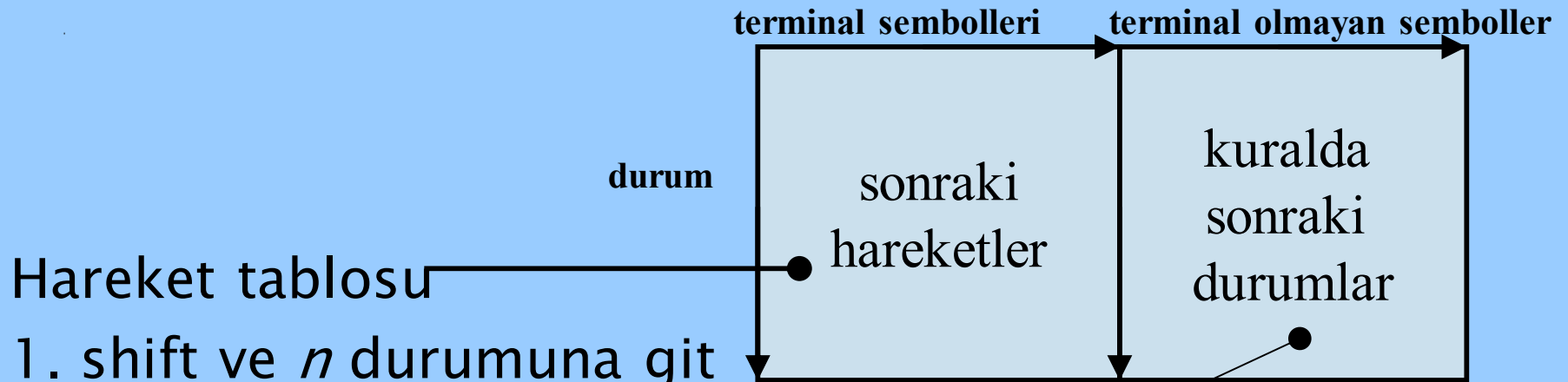
$\xrightarrow{a}$

$X \rightarrow \gamma \blacksquare$

$\xrightarrow{X}$



# Shift-Reduce Ayırıştırma Tablosu



Hareket tablosu

1. shift ve  $n$  durumuna git

2. reduce  $X \rightarrow \gamma$  kullanarak

- $\gamma$  sembolleri yığından al
  - Yığının üstündeki durum etiketini kullanarak  $X$  in goto tablosunda  $X$ 'e bak ve o duruma git
- 
- DFA + yığın = push-down automaton (PDA)

# Liste Gramer Ayırıştırma Tablosu

	(	)	id	,	\$	S	L
1	s3		s2			g4	
2	S→id	S→id	S→id	S→id	S→id		
3	s3		s2			g7	g5
4					accept		
5		s6		s8			
6	S→(L)	S→(L)	S→(L)	S→(L)	S→(L)		
7	L→S	L→S	L→S	L→S	L→S		
8	s3		s2			g9	
9	L→L,S	L→L,S	L→L,S	L→L,S	L→L,S		

# Shift-reduce Ayırıştırma Örneği

- Gramer

$$S' \rightarrow S$$

$$S \rightarrow (S)S \mid \lambda$$

- Ayırıştırma hareketleri

Yığın	Giriş	Hareket
\$	(( )) \$	
\$ (	( ) ) \$	
\$ ( (	) ) \$	
\$ ( ( S	) ) \$	
\$ ( ( S )	) \$	
\$ ( ( S ) S	) \$	
\$ ( S	) \$	
\$ ( S )	\$	
\$ ( S ) S	\$	
\$ S	\$	

- Tersisi

- Sağ türetme

- Soldan sağa

<b>1</b>	$\Rightarrow (( ))$
<b>2</b>	$\Rightarrow ( ( ) )$
<b>3</b>	$\Rightarrow ( ( ) )$
<b>4</b>	$\Rightarrow ( ( S ) )$
<b>5</b>	$\Rightarrow ( ( S ) )$
<b>6</b>	$\Rightarrow ( ( S ) S )$
<b>7</b>	$\Rightarrow ( S )$
<b>8</b>	$\Rightarrow ( S )$
<b>9</b>	$\Rightarrow ( S ) S$
<b>10 S'</b>	$\Rightarrow S$

# Shift-reduce Ayırıştırma Örneği

- Gramer

$S' \rightarrow S$

$S \rightarrow (S)S \mid \lambda$

- Ayırıştırma hareketleri

Yığın	Giriş	Hareket		
\$	(( )) \$	shift	<b>1</b>	$\Rightarrow (( ))$
\$ (	(( )) \$	shift	<b>2</b>	$\Rightarrow ( ( ))$
\$ ( (	(( )) \$	reduce $S \rightarrow \lambda$	<b>3</b>	$\Rightarrow ( ( ))$
\$ ( ( S	(( )) \$	shift	<b>4</b>	$\Rightarrow ( ( S ) )$
\$ ( ( S )	(( )) \$	reduce $S \rightarrow \lambda$	<b>5</b>	$\Rightarrow ( ( S ) )$
\$ ( ( S ) S	(( )) \$	reduce $S \rightarrow ( S ) S$	<b>6</b>	$\Rightarrow ( ( S ) S )$
\$ ( S	(( )) \$	shift	<b>7</b>	$\Rightarrow ( S )$
\$ ( S )	(( )) \$	reduce $S \rightarrow \lambda$	<b>8</b>	$\Rightarrow ( S )$
\$ ( S ) S	(( )) \$	reduce $S \rightarrow ( S ) S$	<b>9</b>	$\Rightarrow ( S ) S$
\$ S	(( )) \$	accept	<b>10</b>	$\Rightarrow S$

**Geçerli prefix** (red arrow pointing to the stack state at step 4: \$ ( ( S ) )\$)

**handle - işleyici** (red arrow pointing to the stack state at step 5: \$ ( ( S ) )\$)

# Shift-Reduce Ayırıştırma

Yığın	Giriş	Hareket
\$	a b b c d e \$	Shift
\$ a	b b c d e \$	Shift
\$ a b	b c d e \$	Reduce A $\Rightarrow$ b
\$ a A	b c d e \$	Shift
\$ a A b	c d e \$	Shift
\$ a A b c	d e \$	Reduce A $\Rightarrow$ A b c
\$ a A	d e \$	Shift
\$ a A d	e \$	Reduce B $\Rightarrow$ d
\$ a A B	e \$	Shift
\$ a A B e	\$	Reduce S $\Rightarrow$ a A B
\$	\$	e

S  $\Rightarrow$  a A B e

A  $\Rightarrow$  A b c | b

B  $\Rightarrow$  d

# Örnek için Ayırıştırma Tablosu

Durum	a	b	c	d	e	\$	S	A	B
0	s1								
1		s3						2	
2		s5		s6					4
3		r3		r3					
4					s7				
5			s8						
6					r4				
7						acc			
8		r2		r2					

1:  $S \Rightarrow a A B e$

2:  $A \Rightarrow A b c$

3:  $A \Rightarrow b$

4:  $B \Rightarrow d$

Action kısmı

Goto kısmı

s, duruma shift anlamına gelir

r bu numaralı kurala  
reduce anlamına gelir

# Terimler

---

- **Right sentential form**
  - Bir sağdan türetmede cümlesel form
- **Geçerli prefix**
  - Ayırıştırma yığının üstündeki semboller sırası
- **İşleyici**
  - Reduction'ın uygulanabildiği + reduction için kuralın kullanıldığı sağ cümlesel form + pozisyon
- **LR(0) item**
  - Sağ tarafında ayrılmış pozisyonlu kural (üretim )
- **Right sentential form**
  - $( S ) S$
  - $(( S ) S )$
- **Viable prefix**
  - $( S ) S, ( S ), ( S, ($
  - $(( S ) S, (( S ), (( S , (( , ($
- **Handle**
  - $( S ) S.$  with  $S \rightarrow \lambda$
  - $( S ) S .$  with  $S \rightarrow \lambda$
  - $(( S ) S . )$  with  $S \rightarrow ( S ) S$
- **LR(0) item**
  - $S \rightarrow ( S ) S.$
  - $S \rightarrow ( S ) . S$
  - $S \rightarrow ( S . ) S$
  - $S \rightarrow ( . S ) S$
  - $S \rightarrow . ( S ) S$

# İtemlerin Sonlu Otomatu (DFA)

Grammer:

$S' \rightarrow S$

$S \rightarrow (S)S$

$S \rightarrow \lambda$

İtemler:

$S' \rightarrow .S$

$S' \rightarrow S.$

$S \rightarrow .(S)S$

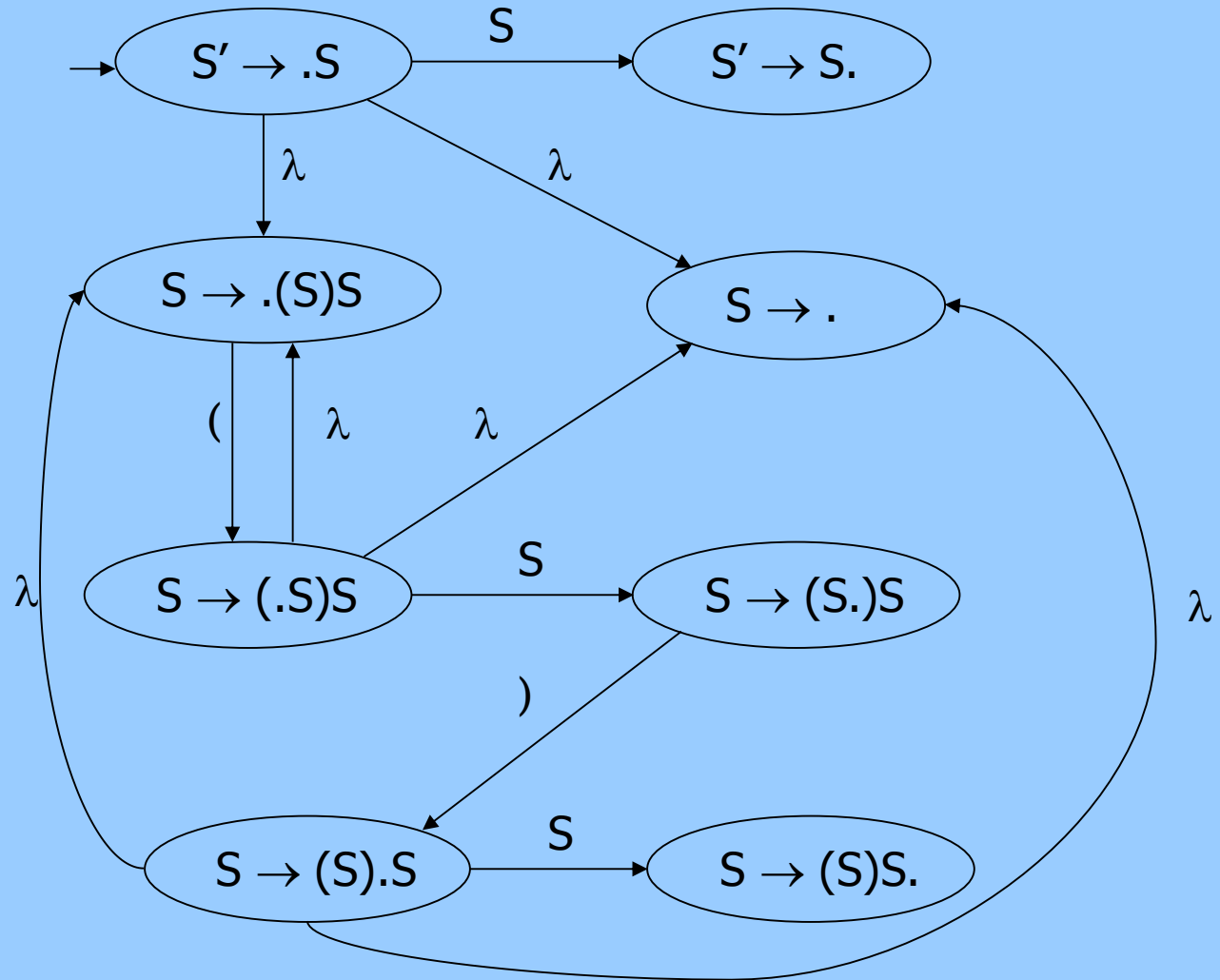
$S \rightarrow (.S)S$

$S \rightarrow (S.)S$

$S \rightarrow (S).S$

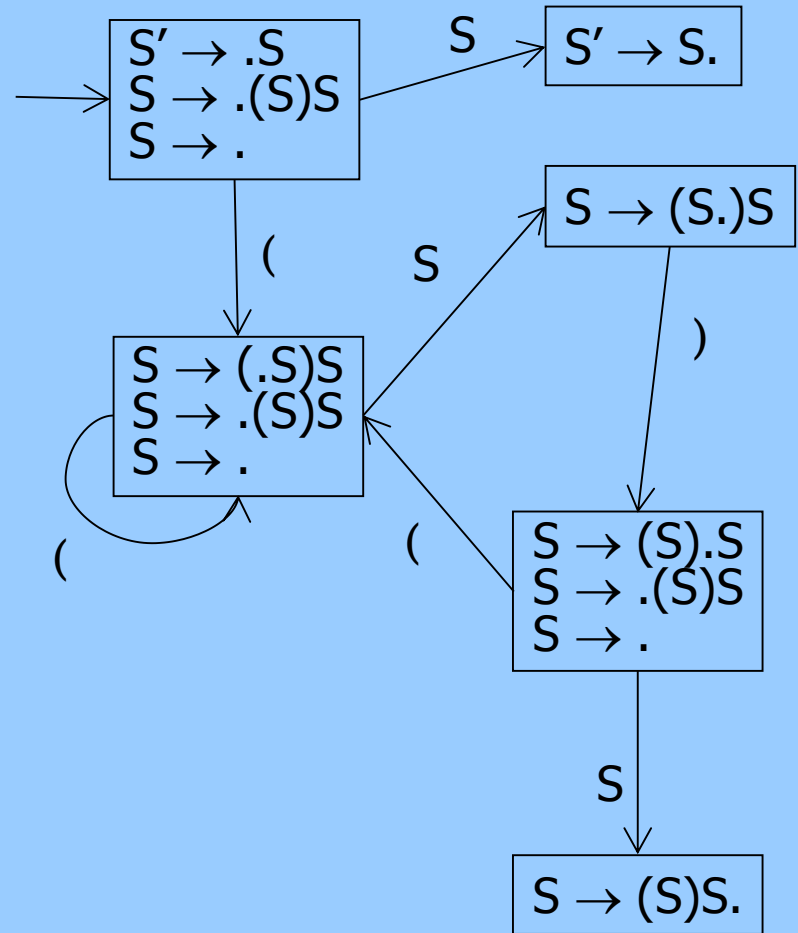
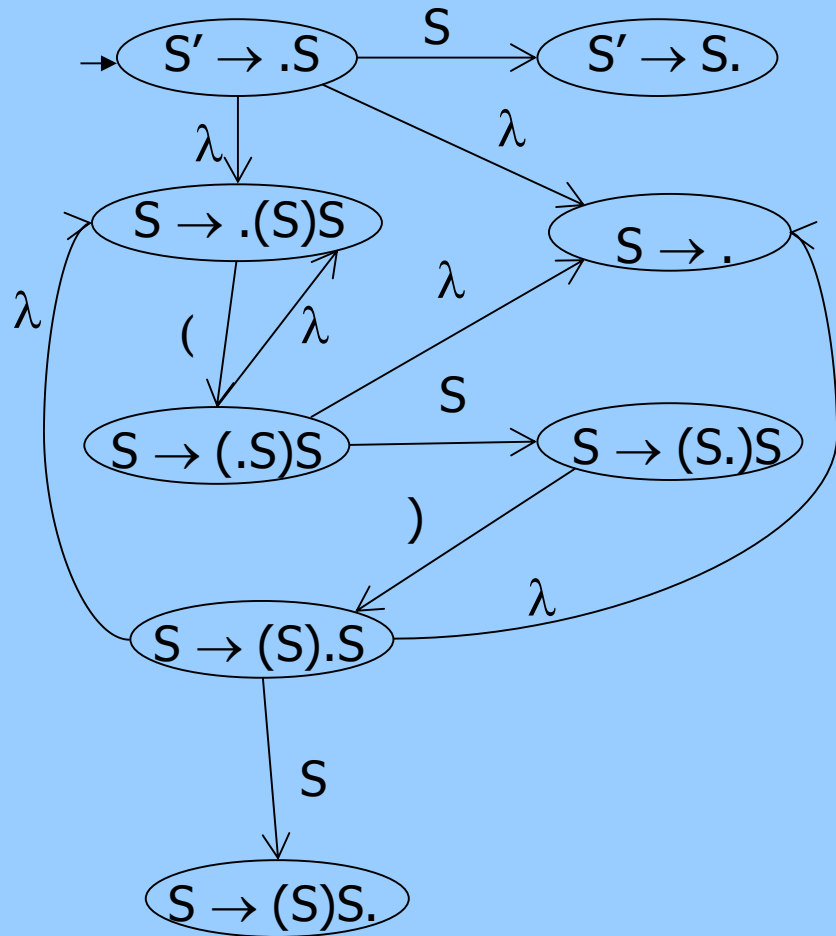
$S \rightarrow (S)S.$

$S \rightarrow .$

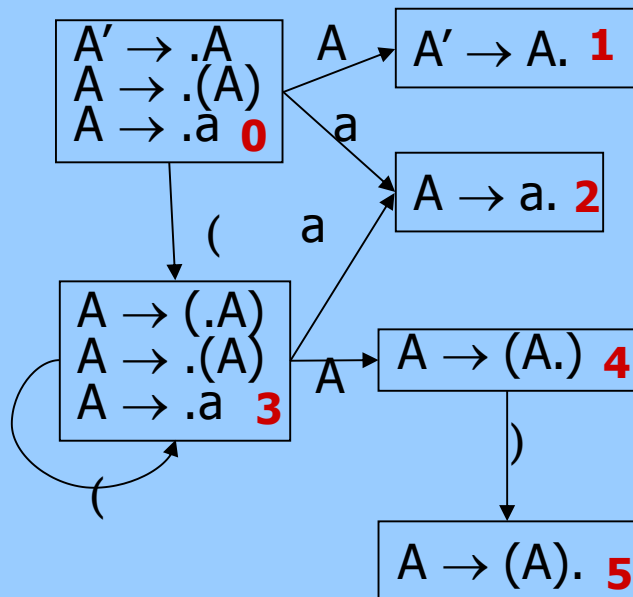




# LR(0) Itemların DFA'sı



# LR(0) Ayırıştırma Tablosu



State	Action	Rule	(	a	)	A
0	shift		3	2		1
1	reduce	$A' \rightarrow A$				
2	reduce	$A \rightarrow a$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \rightarrow (A)$				

# LR(0) Ayırıştırma Örneği

Durum	Hareket	Kural	(	a	)	A
0	shift		3	2		1
1	reduce	$A' \rightarrow A$				
2	reduce	$A \rightarrow a$				
3	shift		3	2		4
4	shift				5	
5	reduce	$A \rightarrow (A)$				

## Yığın

**\$0**  
**\$0(3**  
**\$0(3(3**  
**\$0(3(3a2**  
**\$0(3(3A4**  
**\$0(3(3A4)5**  
**\$0(3A4**  
**\$0(3A4)5**  
**\$0A1**

## Giriş

**((a))\$**  
**(a))\$**  
**a))\$**  
**) )\$**  
**) )\$**  
**)\$**  
**)\$**  
**\$**  
**\$**

## Hareket

**shift**  
**shift**  
**shift**  
**reduce**  
**shift**  
**reduce**  
**shift**  
**reduce**  
**accept**

# LR(0) Sınırlamaları

---

- Bir LR(0) makinesi reduce hareketli durumlar sadece tek reduce hareketine sahipse çalışır – bu durumlarda, bir sonraki bakma ihmal edilip reduce yapılır
- Daha kompleks gramerlerde, yapı shift/reduce ya da reduce/reduce çatışmalı durumlar verir
- Seçmek için look-ahead kullanmaya ihtiyaç duyar

ok

$L \rightarrow L, S.$
-----------------------

shift/reduce

$L \rightarrow L, S.$
$S \rightarrow S., L$

reduce/reduce

$L \rightarrow L, S.$
$L \rightarrow S.$

# LR(1) Ayırıştırma

---

- 1 lookahead sembol ayırıştırma tablosu kadar daha güçlü
- LR(1) grameri = 1 look-ahead'li shift/reduce ayırıştırıcı tarafından tanınabilir.
- LR(1) item = LR(0) item + kuralı takip etmesi mümkün look-ahead sembolleri

LR(0):  $S \rightarrow \cdot S + E$

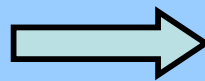
LR(1):  $S \rightarrow \cdot S + E \quad +$

# LR(1) Durum

---

- LR(1) durum = LR(1) itemlar kümesi
- LR(1) item = LR(0) item + lookahead semboller kümesi
- Durumda iki item aynı kurala + nokta konfigürasyonuna sahip olmaz

$S \rightarrow S . + E$	+
$S \rightarrow S . + E$	\$
$S \rightarrow S + . E$	<b>num</b>



$S \rightarrow S . + E$	+, \$
$S \rightarrow S + . E$	<b>num</b>

# LALR Gramerleri

---

- LR(1) ile problem: çok fazla durum
- LALR(1) (Look-Ahead LR)
  - İtemları look-ahead haricinde özdeş iki LR(1) durumunu birleştirir
  - Daha küçük ayrıştırıcı tablo ile sonuçlanır — pratikte oldukça iyi çalışır
  - Otomatik ayrıştırıcı üreticileri için kullanışlı bir teknolojidir

$$\begin{array}{|l} S \rightarrow id . \quad + \\ S \rightarrow E . \quad \$ \end{array} + \begin{array}{|l} S \rightarrow id . \quad \$ \\ S \rightarrow E . \quad + \end{array} = ?$$

# LALR(1) DFA

---

## Algoritma:

repeat

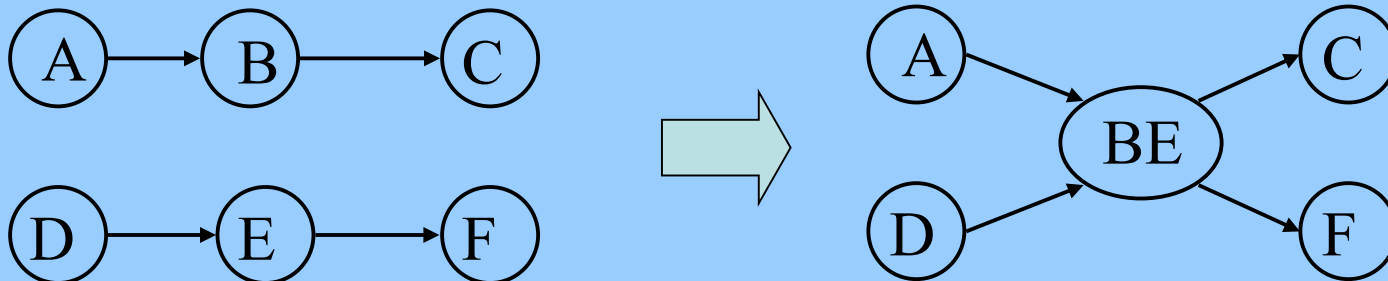
Aynı merkezli iki durumu seç

İtemleri birleştirerek durumları kaynaştır

Atadan yeni duruma ayırıt doğrult

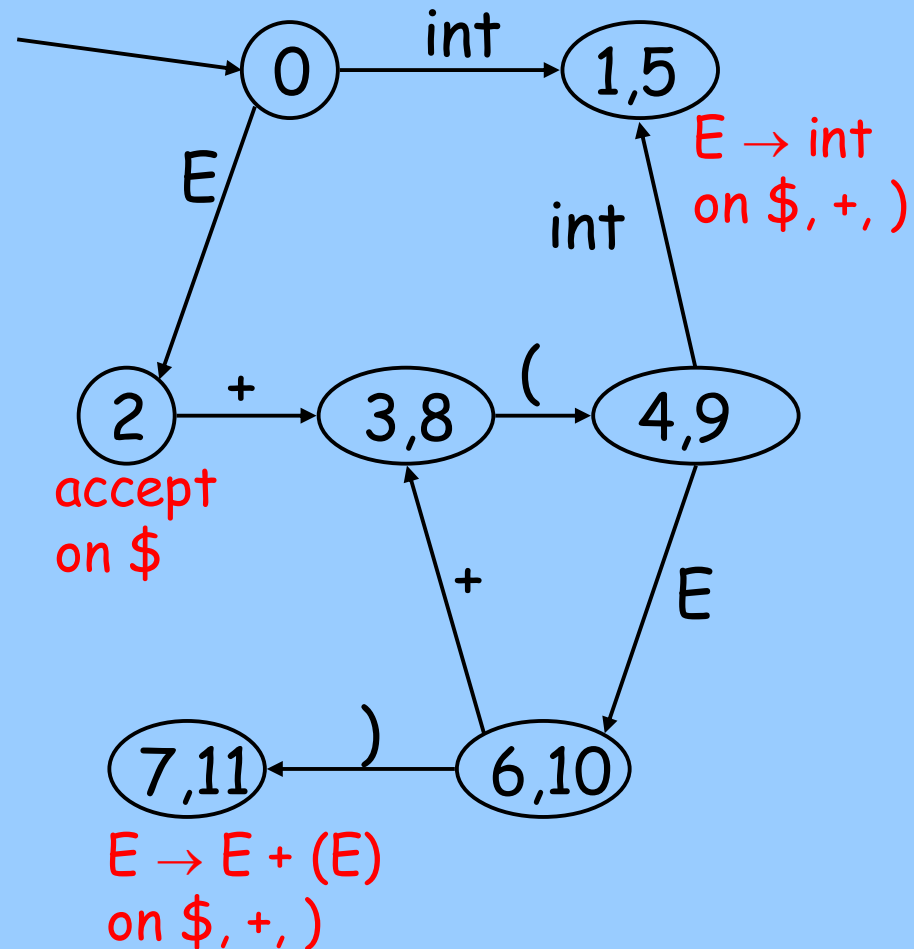
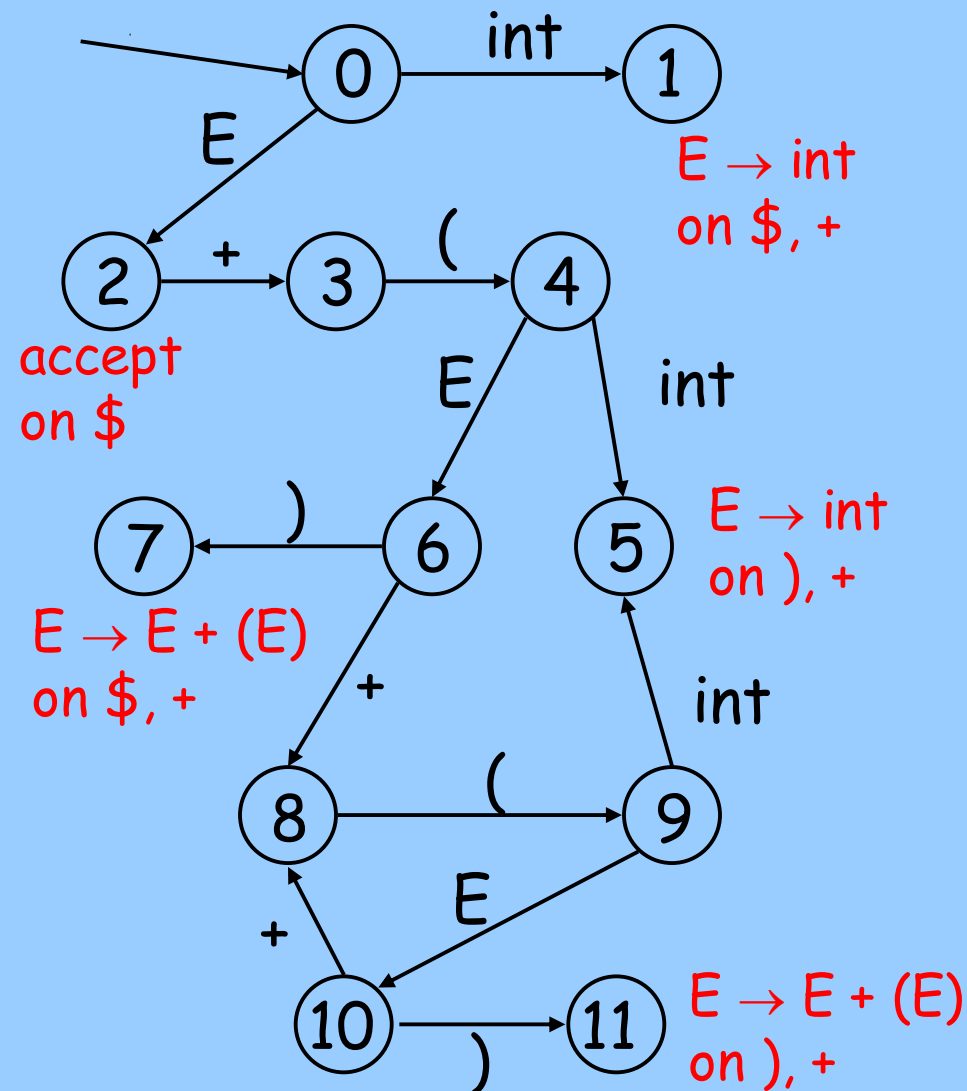
Yeni durum eski tüm ardılları gösterir

until tüm durumlar ayrı merkeze sahip





# LR(1)'i LALR(1)'e Dönüştürme



# Ayrıştırma hakkında notlar

---

## Ayrıştırma

- Sağlam bir temel: içerik–bağımsız gramerler
- Basit bir ayrıştırıcı: LL(1)
- Daha güçlü bir ayrıştırıcı : LR(1)
- Etkili bir vuruş: LALR(1)
  
- LR ayrıştırıcılar ile ilgili sorunlar
- LALR(1) ayrıştırıcı üreticileri

# LR ayrıştırıcılar ile ilgili sorunlar

---

Bir durum şunu içerirse ne olur:

$[X \rightarrow \alpha \cdot \underline{a}\beta, \underline{b}]$  ve  $[Y \rightarrow \gamma \cdot , \underline{a}]$

O zaman “a” girişinde

- $[X \rightarrow \alpha \underline{a} \cdot \beta, \underline{b}]$  duruma **Shift** ,  
ya da
- $Y \rightarrow \gamma$  ile **Reduce**

Bu *shift-reduce çatışması* olarak bilinir

- Belirsizlikten kaynaklanır

# Shift–reduce çatışmaları



Klasik örnek: askıda else

$S \rightarrow \text{if } E \text{ then } S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{OTHER}$

Örnek:

```
if (a == 1) then
  if (b == 4) then
    x = 2;
  else ...    < -- bu hangi 'if'e gider ?
```

Aşağıdakileri içeren DFA durumu olarak

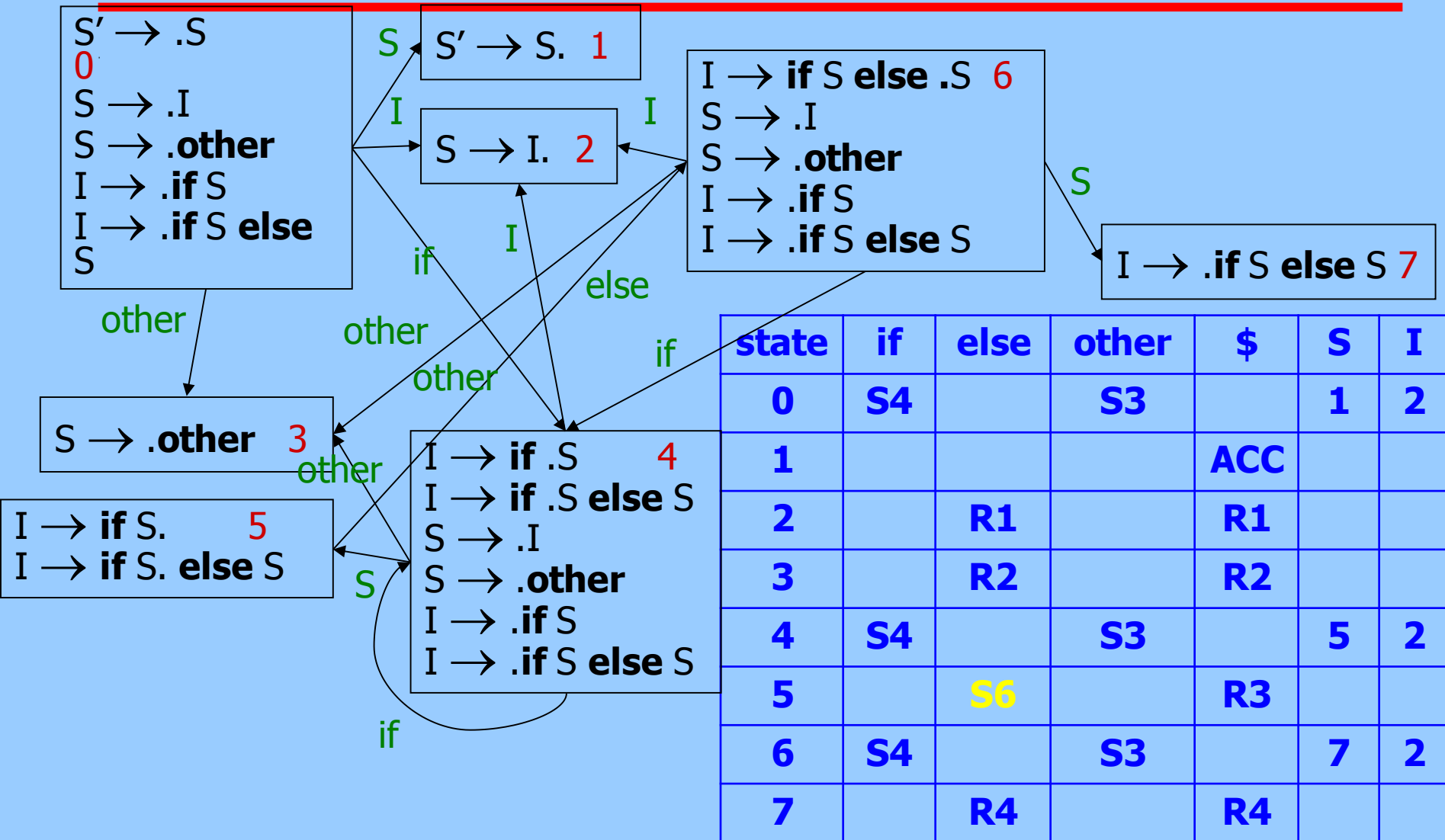
$[S \rightarrow \underline{\text{if}} \ E \ \underline{\text{then}} \ S \ \cdot, \quad \underline{\text{else}}]$

$[S \rightarrow \underline{\text{if}} \ E \ \underline{\text{then}} \ S \ \cdot \ \underline{\text{else}} \ S, \quad \underline{x}]$

Pratik çözümler:

- else'nin önceliğini yansıtacak şekilde grameri düzenle
- Çoğu LR ayrıştırıcılar varsayılan olarak “shift” yapar
- Çoğu kez bir öncelik açıklaması vardır

# Askida Else



# Örnek: Çatışma

---

Meşhur belirsiz gramerimiz:

$$\begin{array}{ccc} E & \rightarrow & E + E \\ & | & E * E \\ & | & (E) \\ & | & \text{int} \end{array}$$

# Shift-Reduce Ayırıştırması

---

| int \* int + int

shift

...

...

E \* E | + int

reduce  $E \rightarrow E * E$

E | + int

shift

E + | int

shift

E + int |

reduce  $E \rightarrow int$

E + E |

reduce  $E \rightarrow E + E$

E |

# Diğer Shift-Reduce Ayırıştırması

---

| int \* int + int

shift

...

...

E \* E | + int

shift

E \* E + | int

shift

E \* E + int |

reduce  $E \rightarrow \text{int}$

E \* E + E |

reduce  $E \rightarrow E + E$

E \* E |

reduce  $E \rightarrow E * E$

E |



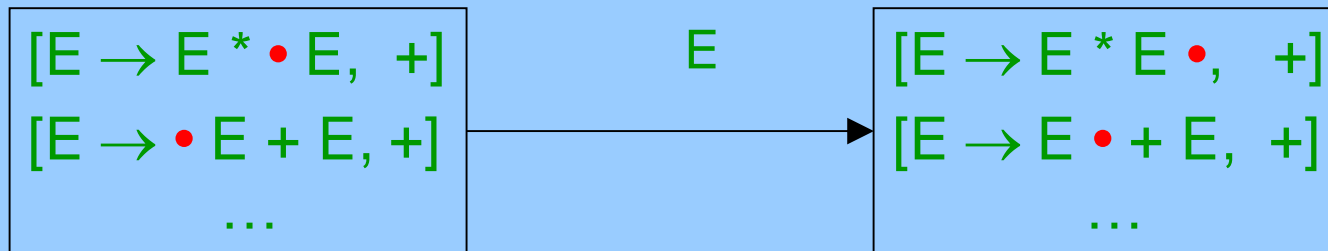
# Örnek

---

Aşağıdaki belirsiz grameri göz önüne alalım

$$E \rightarrow E + E \mid E * E \mid \text{int}$$

DFA kısmı:



+ girişinde bir shift/reduce'a sahibiz

NE yapmak istiyoruz?

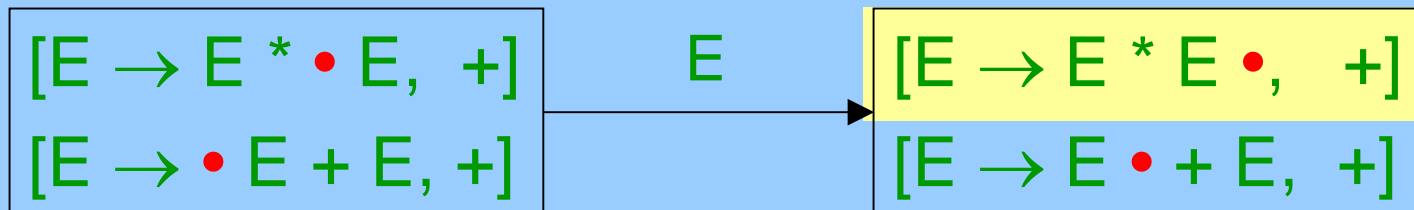
- Düşün:  $x * y + z$
- Reduce'a ihtiyacımız var ( $*$ ,  $+$  dan daha sıkı bağlanır)
- Varsayılan hareket shift'tir

# Öncelik

---

## Görelî önceliği açıkla

- Açıkça çatışmayı önler
- Ayrıştırıcıya anlat: **\*** , **+**'a tercih ederiz



## Pratikte :

- Ayrıştırıcı üreticileri operatörler için öncelik açıklaması desteği verir

# Diğer Problemler

---

Bir DFA durumu ikisini içerirse

$$[X \rightarrow \alpha \cdot, \underline{a}] \text{ ve } [Y \rightarrow \beta \cdot, \underline{a}]$$

- Burada problem nedir?
- Bir sonraki token a olduğunda seçilecek iki reduction vardır

Bu *reduce/reduce* çatışması olarak adlandırılır

- Genellikle gramerde ciddi bir belirsizliktir
- Ayırıştırıcı üretmek için mutlaka düzeltilmelidir

# Reduce/Reduce Çatışmaları

---

Örnek: tanımlayıcılar (identifiers) sırası

$$S \rightarrow \varepsilon \mid id \mid id S$$

**id** stringi için iki ayrıştırma ağacı vardır

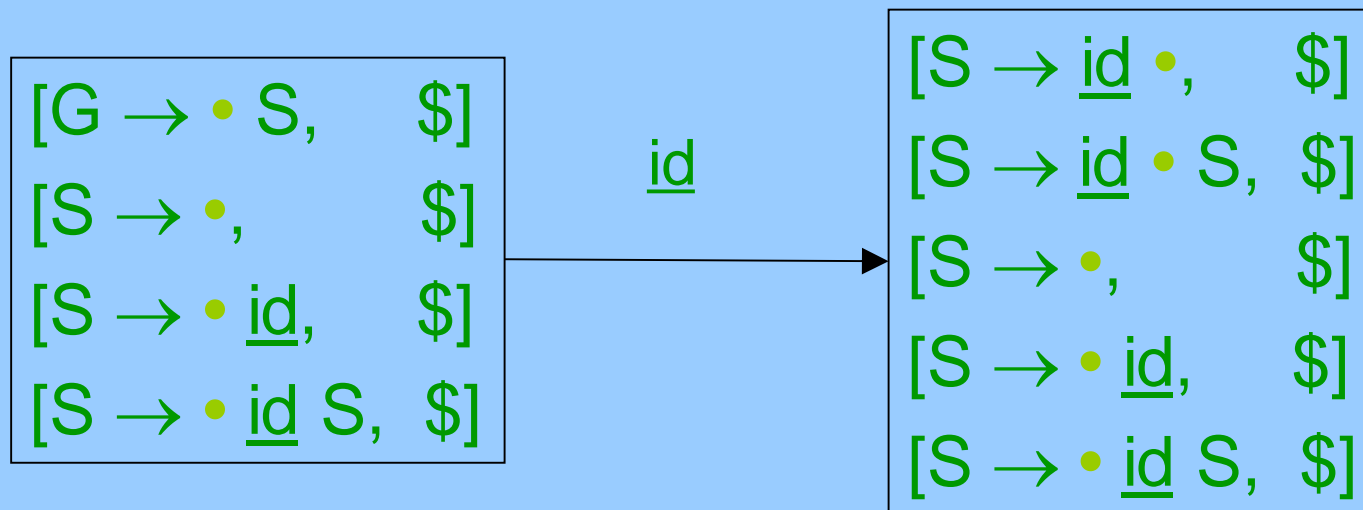
$$S \rightarrow id$$

$$S \rightarrow id S \rightarrow id$$

Bu ayrıştırıcıyı nasıl karıştırır?

# Reduce/Reduce Çatışmaları

DFA durumlarını düşünün:



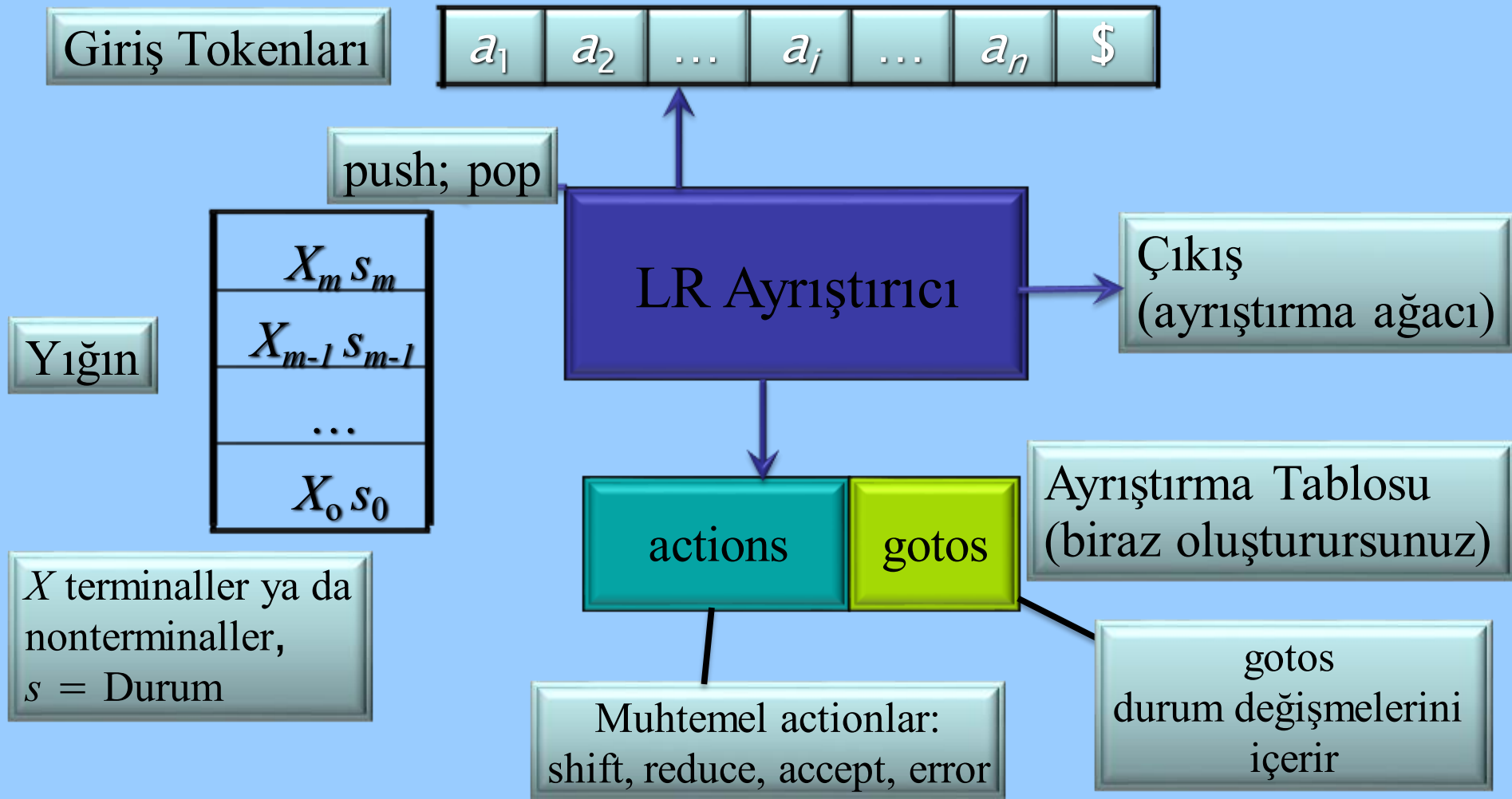
\$ girişinde reduce/reduce çatışması

$G \rightarrow S \rightarrow id$

$G \rightarrow S \rightarrow id S \rightarrow id$

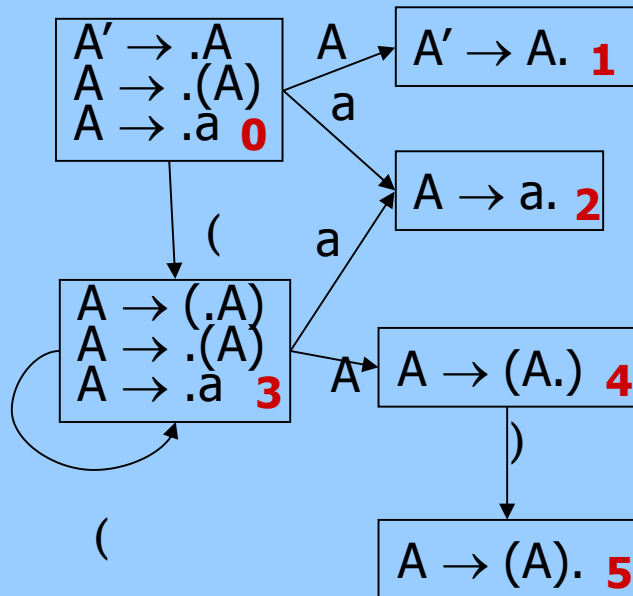
Bunun yerine grameri yediden yazarız:  $S \rightarrow \varepsilon \mid id S$

# Bir LR Ayırıştırıcı



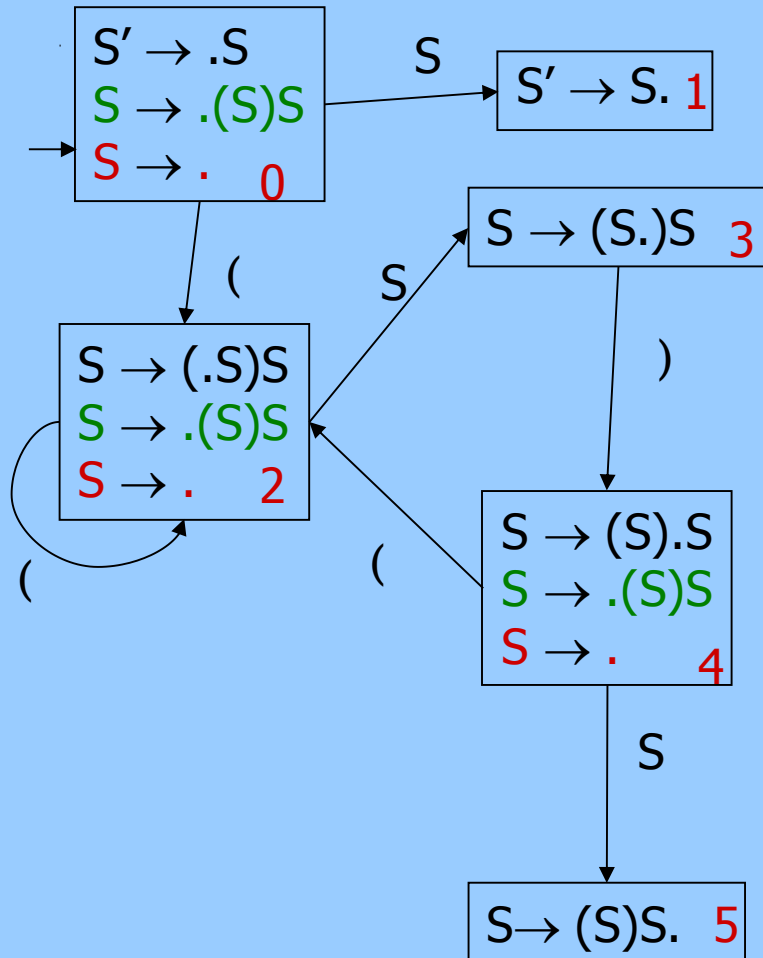
# SLR(1) Ayırıştırma Tablosu

$A \rightarrow (A) \mid a$



State	(	a	)	\$	A
0	S3	S2			1
1				AC	
2			R2		
3	S3	S2			4
4			S5		
5			R1		

# LR(0) olmayan SLR(1) Gramer



$S \rightarrow (S)S \mid \lambda$

State	(	)	\$	S
0	S2	R2	R2	1
1			AC	
2	S2	R2	R2	3
3		S4		
4	S2	R2	R2	5
5		R1	R1	



## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

İşleyiciler (tanıtıcı değer) (handles) hakkında:

- Tanım:  $\beta$  , sağ cümlesel formun işleyicisidir  
(tanıtıcı değeridir) (handle)

$\gamma = \alpha\beta w$  ancak ve ancak

$$S \Rightarrow^* \alpha A w \Rightarrow \alpha \beta w$$

- Tanım:  $\beta$  , sağ cümlesel formun tümceciğidir  
 $\gamma$  ancak ve ancak

$$S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$$

- Tanım:  $\beta$  , sağ cümlesel form  $\gamma$  nın basit  
tümceciğidir ancak ve ancak

$$S \Rightarrow^* \gamma = \alpha_1 A \alpha_2 \Rightarrow \alpha_1 \beta \alpha_2$$

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- İşleyiciler (tanıtıcı değer) (handles) hakkında :
  - Bir sağ cümlesel formun işleyicisi onun en soldaki basit tümceciğidir
  - Verilen bir ayırıştırma ağacında, şimdi işleyiciyi (handle) bulmak kolaydır
  - Ayırıştırma, işleyici budama (handle pruning) olarak düşünülebilir

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- **Kaydırma–İndirgeme Algoritmaları** (Shift–Reduce Algorithms)
  - İndirgeme (Reduce), ayırıştırma yığınının üstündeki işleyici (handle) ile ona ilişkin LHS'nin yerini değiştirme işlemidir
  - Kaydırma (Shift), bir sonraki jetonu ayırıştırma yığınının üstüne koyma işlemidir

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- LR ayırıştırıcıların avantajları:
  - Programlama dillerini tanımlayan gramerlerin hemen hemen tümü için çalışır.
  - Diğer aşağıdan–yukarıya algoritmalarından daha geniş bir gramerler sınıfı için çalışır, fakat diğer aşağıdan–yukarıya ayırıştırıcıların herhangi biri kadar da verimlidir
  - Mümkün olan en kısa zamanda sentaks hatalarını saptayabilir
  - LR gramerler sınıfı, LL ayırıştırıcıları tarafından ayrıştırılabilen sınıfın üstkümesidir (superset)

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- LR ayırıştırıcıları bir araç ile oluşturulmalıdır
- Knuth'un görüşü: Bir aşağıdan–yukarıya ayırıştırıcı, ayırıştırma kararları almak için, ayırıştırmanın o ana kadar olan bütün geçmişini kullanabilir
  - Sonlu ve nispeten az sayıda farklı ayırıştırma durumu oluşabilir, bu yüzden geçmiş ayırıştırma yığını üzerinde bir ayırıştırıcı durumunda saklanabilir

## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- Bir LR yapılandırması bir LR ayırıştırıcının durumunu saklar
- $(S_0X_1S_1X_2S_2\ldots X_mS_m, a_ia_{i+1}\ldots a_n\$)$

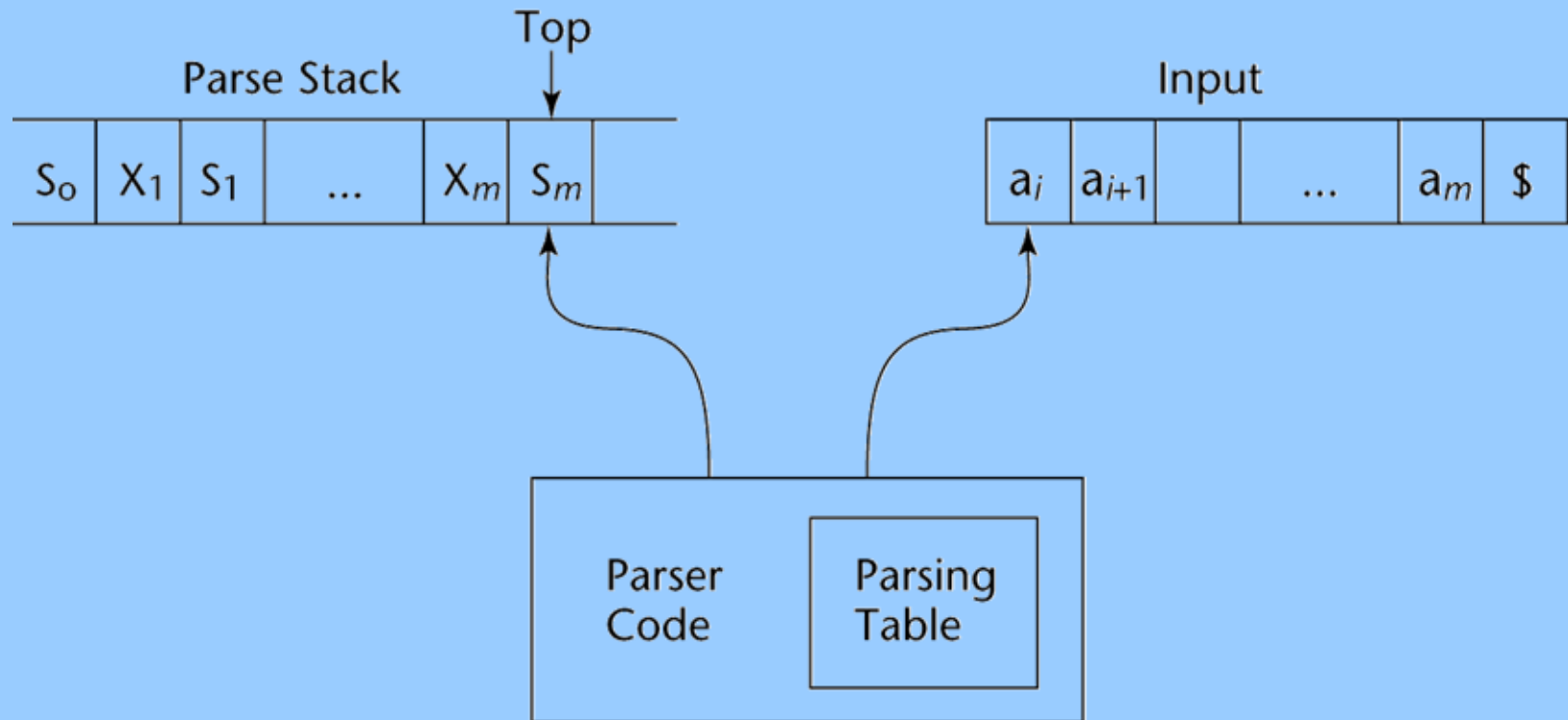
## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- LR ayırıştırıcılar tablo sürümlüdür, bu tablonun iki bileşeni vardır, bir ACTION tablosu ve bir GOTO tablosu
  - ACTION tablosu, verilen bir ayırıştırıcı durumu ve sonraki jeton için, ayırıştırıcının hareketini belirler
    - Satırlar durum adlarıdır; sütunlar terminallerdir
  - The GOTO tablosu bir indirgeme hareketi yapıldıktan sonra ayırıştırma yığını üzerine hangi durumun konulacağını belirler
    - Satırlar durum adlarıdır; sütunlar nonterminallerdir

# Bir LR Ayırıştırıcısının (Parser) Yapısı

---





## 4.5 Aşağıdan–Yukarıya Ayırıştırma (Bottom–up Parsing) (Devam)

---

- Başlangıç yapılandırması:  $(S_0, a_1 \dots a_n \$)$
- Ayırıştırıcı hareketleri:
  - If  $\text{ACTION}[S_m, a_i] = \text{Shift } S$ , sonraki yapılandırma:  
 $(S_0 X_1 S_1 X_2 S_2 \dots X_m S_m a_i S, a_{i+1} \dots a_n \$)$
  - If  $\text{ACTION}[S_m, a_i] = \text{Reduce } A \rightarrow \beta$  ve  $S = \text{GOTO}[S_{m-r}, A]$ ,  $r = \beta$  nın uzunluğu olmak üzere , sonraki yapılandırma:  
 $(S_0 X_1 S_1 X_2 S_2 \dots X_{m-r} S_{m-r} AS, a_i a_{i+1} \dots a_n \$)$

## 4.5 Aşağıdan-Yukarıya Ayırıştırma (Bottom-up Parsing) (Devam)

---

- Ayırıştırıcı hareketleri (devamı):
  - If ACTION[Sm, ai] = Accept, ayırıştırma tamamlanmıştır ve hata bulunmamıştır
  - If ACTION[Sm, ai] = Error, ayırıştırıcı bir hata-işleme rutini çağırır

## 4.5 Aşağıdan-Yukarıya Ayırıştırma Örnek

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

$E \rightarrow E+T$   
 $\quad \quad | T$   
 $\quad \quad | E-T$   
 $T \rightarrow T*F$   
 $\quad \quad | F$   
 $\quad \quad | T/F$   
 $F \rightarrow (E)$   
 $\quad \quad | id$   
 $\quad \quad | -E$   
 $\quad \quad num$

# LR Ayırıştırma Tablosu(Parsing Table)

	Action						Goto		
State	id	+	*	(	)	\$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

## 4.5 Aşağıdan-Yukarıya Ayırıştırma (Bottom-up Parsing) (Devam)

---

- Verilen bir gramerden bir araç ile bir ayırıştırıcı tablosu üretilebilir, örn., **yacc**

# Özet

---

- Sentaks analizi dil implementasyonunun ortak kısmıdır
- Bir sözcüksel analizci bir programın küçük-ölçekli parçalarını ayıran bir desen eşleştiricidir
  - Sentaks hatalarını saptar
  - Bir ayrıştırma ağacı üretir
- Bir özyineli-iniş ayrıştırıcı bir LL ayrıştırıcıdır
  - EBNF
- Aşağıdan-yukarıya ayrıştırıcıların ayrıştırma problemi: o anki cümlesel formun altstringini bulma
- LR ailesi kaydırma-indirgeme ayrıştırıcıları en yaygın olan aşağıdan-yukarıya ayrıştırma yaklaşımıdır

# Kaynaklar

---

- Roberto Sebesta, Concepts Of Programming Languages, International 10th Edition 2013
- David Watt, Programming Language Design Concepts, 2004
- Michael Scott, Programming Languages Pragmatics, Third Edition, 2009
- Zeynep Orhan, Programlama Dilleri Ders Notları
- Mustafa Şahin, Programlama Dilleri Ders Notları
- Ahmet Yesevi Üniversitesi, Uzaktan Eğitim Notları
- Erkan Tanyıldızı, Programlama Dilleri Ders Notları
- Tuğrul Yılmaz, Programlama Dilleri Ders Notları
- David Evans, Programming Languages Lecture Notes
- O. Nierstrasz, Programming Languages Lecture Notes
- Rance Cleaveland and Jeff Foster, Organization of Programming Languages Lecture Notes