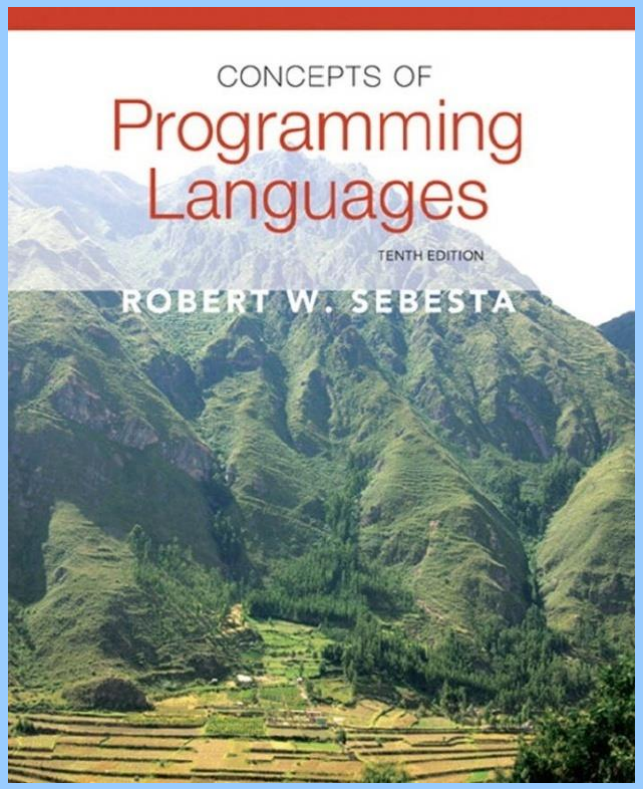
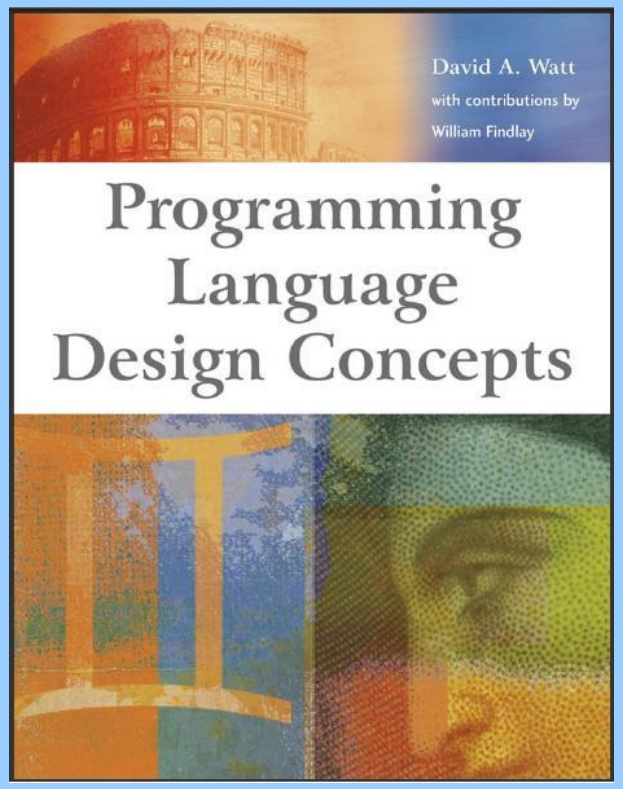
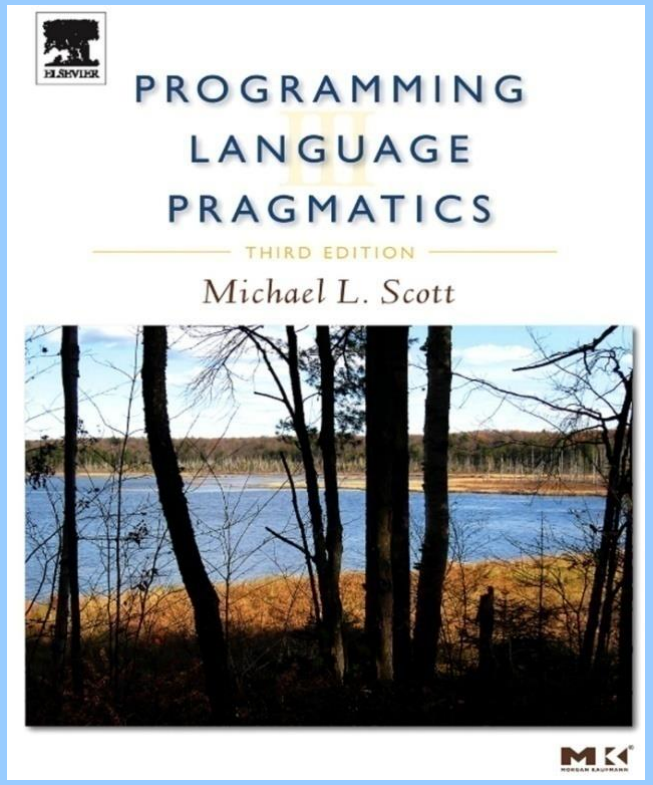
Bölüm 4: Sözcüksel (Lexical) ve Sentaks Analiz







Bölüm 4 Konular

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

4.1 Giriş

- Dil implementasyon sistemleri, belirli implementasyon yaklaşımına aldırmadan 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
  - Ayrıştırıcı doğrudan BNF ye dayalı olabilir
  - BNF ye dayalı ayrıştırıcıların 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 taşınabilirdir

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
  - Lexemeler, jeton (token) adı verilen sözcüksel (lexical) bir kategoriyle ilişkilendirilmiş olan bir karakter desenini eşleştirir
  - **`sum`** bir lexemedir; 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 tanıtıcıyı (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

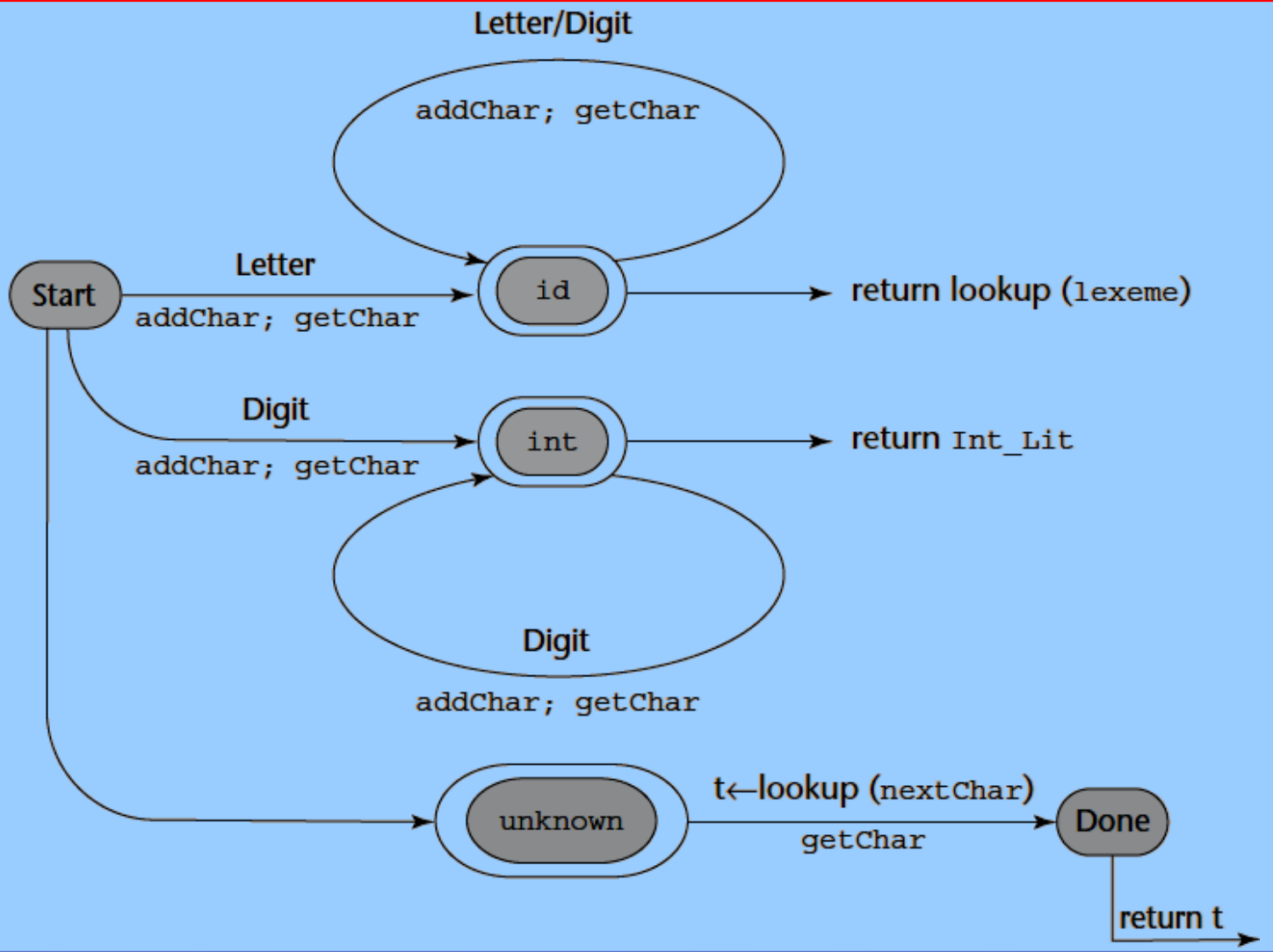
4.2 Sözcüksel (Lexical) Analiz (Devamı)

- 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 lexemenin 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.
```

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
:=	i	-	i	;	writeln	(	i
)	**end**	.					

4.3 Ayrıştırma (Parsing) Problemi

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

4.3 Ayrıştırma (Parsing) Problemi (Devamı)

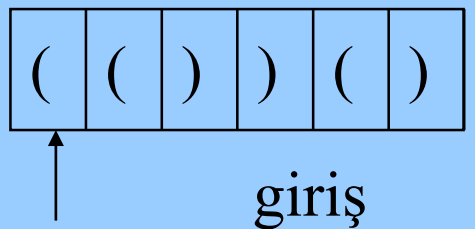
- Ayrıştırıcıların iki kategorisi:
  - Yukarıdan-aşağıya (Top down) – ayrıştırma ağacını kökten başlayarak oluşturur
    - Ayrıştırma ağacını preorderda izler veya oluşturur
  - Aşağıdan-yukarıya (Bottom up) – ayrıştırma ağacını, yapraklardan başlayarak oluşturur
- Ayrıştırıcılar, girdide sadece bir jeton (token) ileriye bakar

4.3 Ayrıştırma (Parsing) Problemi (Devamı)

- Yukarıdan-aşağıya ayrıştırıcılar (Top-down parsers)
  - Bir $xA\alpha$ sağ cümlesel formu (right sentential form) verildiğinde , ayrıştırıcı, 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 ayrıştırma (top-down parsing) algoritmaları:
  - Özyineli azalan (recursive-descent)- kodlanmış bir implementasyon
  - LL ayrıştırıcılar (parser) – tablo sürümlü implementasyon

(())() için Leftmost Türetme

L	// L => (L) L
⇒ (L) L	// L => (L) L
⇒ ((L) L) L	// L => ε
⇒ (( ) L) L	// L => ε
⇒ (( )) L	// L => ε
⇒ (( )) (L) L	// L =>(L) L
⇒ (( )) ( ) L	// L => ε
⇒ (( )) ( )	



4.3 Ayrıştırma (Parsing) Problemi (Devamı)

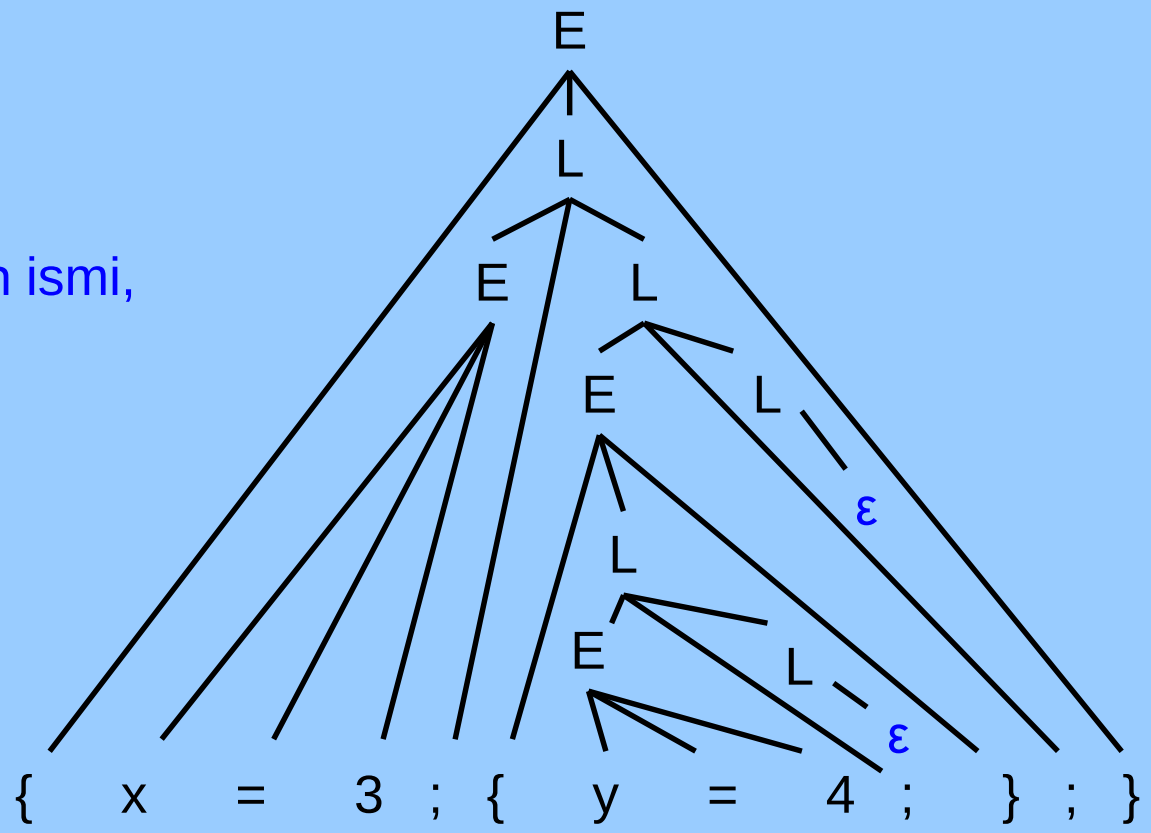
Yukarıdan-aşağıya ayrıştırıcılar (Top-down parsers)

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

$L \rightarrow E ; L \mid \varepsilon$

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

{ x = 3 ; { y = 4 ; } ; }
için ayrıştırma
ağacını göster



4.3 Ayrıştırma (Parsing) Problemi (Devamı)

- Aşağıdan-yukarıya ayrıştırıcılar (bottom-up parsers)
  - Bir α sağ cümlesel formu (right sentential form) verildiğinde, α 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 ayrıştırma algoritmaları LR ailesindedir

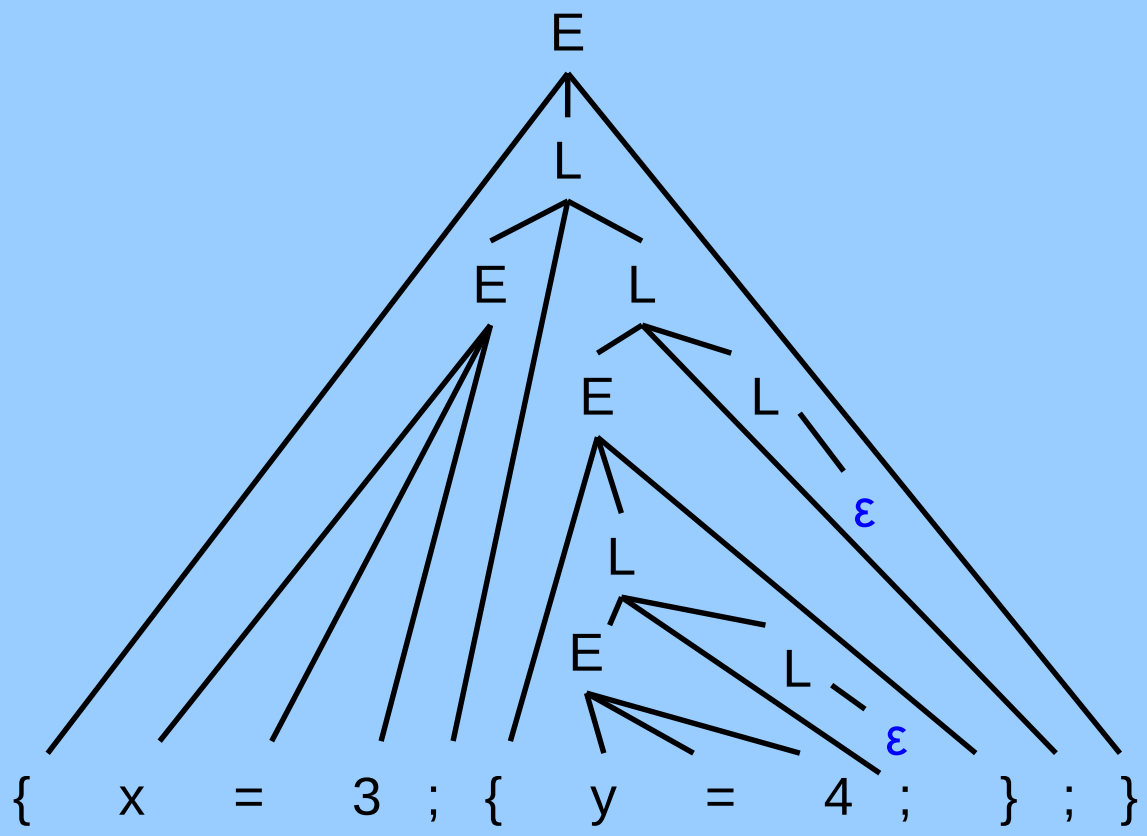
4.3 Ayrıştırma (Parsing) Problemi (Devamı)

Aşağıdan-yukarıya ayrıştırıcılar (bottom-up parsers)

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

{ x = 3 ; { y = 4 ; } ; }
için ayrıştırma
ağacını göster

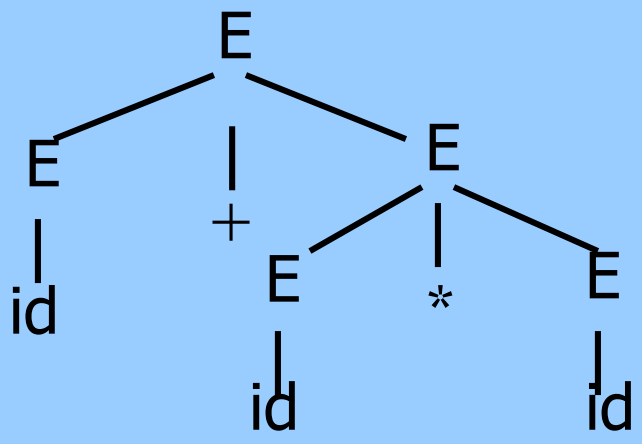
Oluşturulan sonuç ağaçlar yukarıdan-aşağıya ile aynıdır; sadece düğümlerin ağaca eklenme sırası değişiktir



4.3 Ayrıştırma (Parsing) Problemi (Devamı)

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

Ayrıştırma Ağaçları ve Türetmeler



Top-down ayrıştırma

$$
\begin{aligned}
E &\Rightarrow E + E \\
&\Rightarrow id + E \\
&\Rightarrow id + E * E \\
&\Rightarrow id + id * E \\
&\Rightarrow id + id * id
\end{aligned}
$$

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)

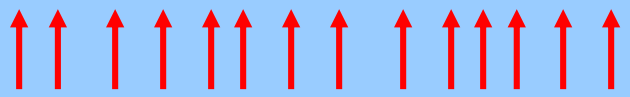
- Özyineli-azalan işlem (Recursive-descent Process) (Yukarıdan-Aşağıya ayrıştırma yapar)
  - Gramerde her bir nonterminal için o nonterminal tarafından üretilebilen cümleleri ayrıştırabilen bir altprogram vardır
  - EBNF, özyineli-azalan ayrıştırıcıya (recursive-descent parser) temel oluşturmak için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

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)

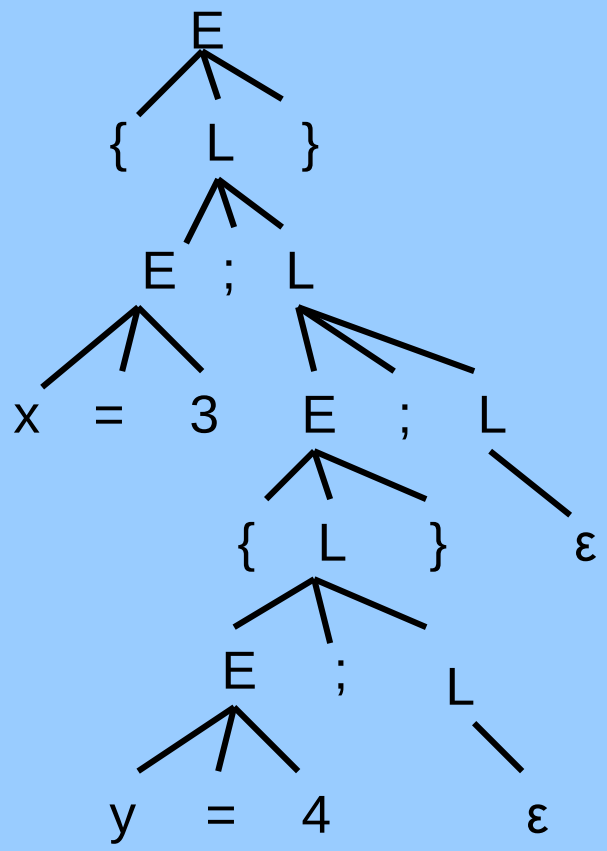
E → id = n | { L }

L → E ; L | ϵ

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



lookahead



4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing) (Devamı)

- Basit deyimler (expressions) için bir gramer:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | ( <expr> )
```

4.4 Özyineli-azalan Ayrıştırma (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 ayrıştırıcı alt programını çağırır

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

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

void expr() {

/* İlk terimiayrıştır*/

 term();
  ...
```

4.4 Özyineli-azalan Ayrıştırma(Recursive-Descent Parsing)(Devamı)

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

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

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

4.4 Özyineli-azalan Ayrıştırma (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 Ayrıştırma (Recursive-Descent Parsing)(Devamı)

```
/* factor fonksiyonu dilde şu kuralın
  ürettiği stringleri ayrıştırır:
   <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 Ayrıştırma (Recursive-Descent Parsing)(Devamı)

```
/* Eğer sagdaki 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 Ayrıştırma (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) ayrıştırıcının temeli olamaz
      - Bir gramer, sol özyinelemeyi yok etmek için değiştirilebilir

4.4 Özyineli-azalan Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- Yukarıdan-aşağıya ayrıştırmaya izin vermeyen gramerlerin diğer bir özelliği pairwise disjointness (çiftli ayrıklık) eksikliğidir
  - Doğru olan sağ kısmı (RHS) lookaheadın bir jetonuna dayanarak belirleyememesi
  - Tanım: FIRST(α) = {a | α =>* aβ }

    (Eğer α =>* ε ise, ε FIRST(α) içindedir)

  (First(X) herhangi bir cümlesel formda X'ten türetilen ilk terminal kümesidir)

  α =>* : sıfır veya daha fazla türetme varsa

4.4 Özyineli-azalan Ayrıştırma (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 Ayrıştırma (Recursive-Descent Parsing)(Devamı)

- Sol çarpan alma (Left factoring) problemi çözebilir

  Şu ifadeyi:

<variable> → identifier | identifier [<expression>]

  aşağıdakilerden biriyle değiştirin:

<variable> → identifier <new>

<new> → ε | [<expression>]

  veya

<variable> → identifier [[<expression>]]

(dıştaki köşeli parantezler EBNF'nin metasembolleridir)

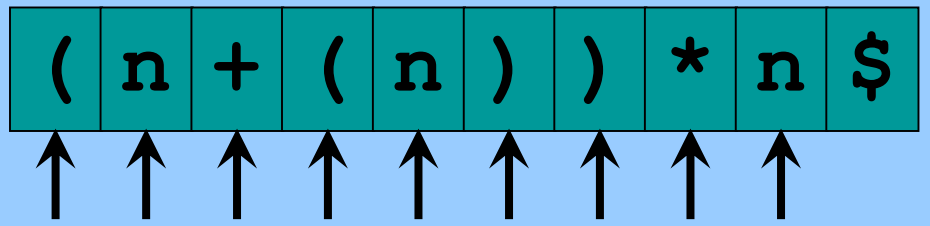
Özyineli-azalan Ayrıştırmada Problemler

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

LL(1) Ayrıştırma Örneği

E ⇒TX
⇒FNX
⇒(E)NX
⇒(TX)NX
⇒(FNX)NX
⇒(nNX)NX
⇒(nX)NX
⇒(nATX)NX
⇒(n+TX)NX
⇒(n+FNX)NX
⇒(n+(E)NX)NX
⇒(n+(TX)NX)NX
⇒(n+(FNX)NX)NX
⇒(n+(nNX)NX)NX
⇒(n+(nX)NX)NX
⇒(n+(n)NX)NX
⇒(n+(n)X)NX
⇒(n+(n))NX
⇒(n+(n))MFNX
⇒(n+(n))*FNX
⇒(n+(n))*nNX
⇒(n+(n))*nX
⇒(n+(n))*n

Bitti



E → T X
X → A T X | λ
A → + | -
T → F N
N → M F N | λ
M → *
F → (E) | n

LL(1) Ayrıştırma Algoritması

Giriş sembolünü 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 ayrıştırma tablosu girişi M[A, a] boş değil THEN

ayrıştırma tablosu girişi M[A, a] dan A $\rightarrow X_1\ X_2\ \ldots\ X_n$ al

Pop yığın;

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

ELSE Error

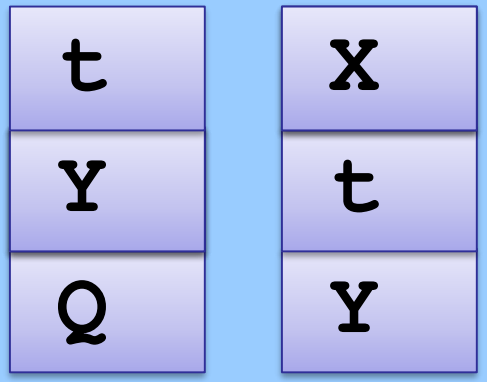
CASE ($,$): Accept

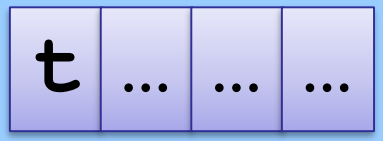
OTHER: Error

LL(1) Ayrıştırma 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 $S \Rightarrow^* WNtY$





First Kümesi

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

First Küme Örnekleri

exp → exp addop term | term
addop → + | –
term → term mulop factor | factor
mulop → *
factor → (exp) | num

First(addop) = {+, –}
First(mulop) = {*}
First(factor) = {(, num}
First(term) = {(, num}
First(exp) = {(, num}

st → ifst | other
ifst → if (exp) st elsepart
elsepart → else st | λ
exp → 0 | 1

First(exp) = {0,1}
First(elsepart) = {else, λ}
First(ifst) = {if}
First(st) = {if, other}

First(A) Bulma Algoritması

For tüm a terminalleri, First(a) = {a}

For tüm A nonterminalleri, First(A) := {}

While herhangi First(A)'ya değişiklik var

  For her kural $A \rightarrow X_1\ X_2 \ldots X_n$

    For $\{X_1, X_2, \ldots, X_n\}$'de her X_i

      If for tüm $j<i$ $First(X_j)$ λ içerir,

      Then

        $First(X_i)-\{\lambda\}$'yı First(A)'ya ekle

  If λ; $First(X_1)$, $First(X_2)$, …, ve $First(X_n)$'de

  Then λ'yı First(A)'ya ekle

If A bir terminal ya da λ, then First(A) = {A}.

If A bir nonterminal, then for her kural $A \rightarrow X_1\ X_2 \ldots X_n$, First(A) $First(X_1) - \{\lambda\}$'yı içerir.

If aynı zamanda for bazı $i<n$, $First(X_1)$, $First(X_2)$, …, ve $First(X_i)$ λ'yı içerir, then First(A) $First(X_{i+1})-\{\lambda\}$'yı içerir.

If $First(X_1)$, $First(X_2)$, …, ve $First(X_n)$ λ'yı içerir, then First(A) da λ 'yı içerir.

First Küme Bulma: Bir Örnek

exp → term exp'

exp' → addop term exp' | λ

addop → + | –

term → factor term'

term' → mulop factor term'
| λ

mulop → *

factor → (exp) | num

	First
exp	
exp'	λ
addop	+ –
term	(num
term'	λ
mulop	*
factor	(num

Follow Kümesi

- $ giriş tokenlarının sonunu göstersin
- A başlangıç sembolü ise, o zaman $, Follow(A)'dadır.
- Bir B $\rightarrow$ X A Y kuralı varsa, o zaman First(Y) – {λ} Follow(A)'dadır.
- B $\rightarrow$ X A Y üretimi (kuralı) varsa ve λ, First(Y)'de ise, o zaman Follow(A), 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 $A \rightarrow X_1\ X_2 \ldots X_n$,

FOR each nonterminal X_i

First($X_{i+1}\ X_{i+2} \ldots X_n$)–{λ} 'yi Follow($X_i$)'e ekle.

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

IF λ First($X_{i+1}\ X_{i+2} \ldots X_n$)'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) – {λ} Follow(X)'tedir.

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

First ve Follow Örnekleri

Gramer	First	Follow
S→ABCDE	{a,b,c}	{$}
A→a/λ	{a, λ}	{b,c}
B→b/λ	{b, λ}	{c}
C→c	{c}	{d,e,$}
D→d/λ	{d, λ}	{e,$}
E→e/λ	{e, λ}	{$}

Gramer	First	Follow
S→Bb/Cd	{a,b,c,d}	{$}
B→aB/λ	{a, λ}	{b}
C→cC/λ	{c, λ}	{d}

Gramer	First	Follow
E→TE'	{id,(}	{$,)}
E'→+TE'/λ	{+, λ}	{$,)}
T→FT'	{id,(}	{+,$,)}
T'→*FT'/λ	{*, λ}	{+,$,)}
F→id/(E)	{id,(}	{*,+,$)}

Follow Kümesi Bulma: Bir Örnek

exp → term exp'

exp' → addop term exp' | λ

addop → + | -

term → factor term'

term' → mulop factor term'
|λ

mulop → *

factor → (exp) | num

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

LL Ayrıştırma Tablosu Oluşturma

Gramer	First	Follow
E→TE'	{id,(}	{$,)}
E'→+TE'/λ	{+, λ}	{$,)}
T→FT'	{id,(}	{+,$,)}
T'→*FT'/λ	{*, λ}	{+,$,)}
F→id/(E)	{id,(}	{*,+,$)}

- E satırında iken E nin first kümesi ilk türetilecek olanlardır. Dolayısıyla id'ye ve ('ya bu kural yazılır.
- E' satırında iken first kümesi + ve λ dır. + sütununa E'→+TE' yazılır. Fakat λ nın firsti yoktur. Dolayısıyla E' ifadesinin follow'una E→ λ deyimi eklenir.

	İd	+	*	(	)	$
E	E→TE'			E→TE'		
E'		E'→+TE'			E'→λ	E'→λ
T	T→FT'			T→FT'		
T'		T'→λ	T'→*FT'		T'→λ	T'→λ
F	F→id			F→(E)		

Örnek: LL ayrıştırma örnek

- Gramer: S→(S)|λ
- Giriş Stringi (())$

	First	Follow
S→(S)|λ	{(, λ }	{$,)}

LL ayrıştırma tablosu

	(	)	$
S	S→(S)	S→λ	S→λ

(	(	)	)	$
X				

STACK
S
$

S→(S)

(
S
)
$

(	(	)	)	$
	X			

Stack'in üstünde (ve girişte (var. Stack'ten (sil.

S
)
$

S→(S)

(
S
)
)
$

(	(	)	)	$
	X			

Stack'in üstünde (ve girişte (var. Stack'ten (sil.

Örnek: LL ayrıştırma devam

(	(	)	)	$
		X		

(	(	)	)	$
		X		

(	(	)	)	$
			X	

S
)
)
$

Stack'in üstünde S var ve giriş) ise S→λ yaz.

)
)
$

Stack'in üstünde) var ve giriş) ise) stack'ten sil ve ilerle.

)
$

Stack'in üstünde) var ve giriş) ise) stack'ten sil ve ilerle.

$

(	(	)	)	$
				X

Stack'in üstünde $ var ve giriş $ ise KABUL

Örnek: LL(1) Ayrıştırma Tabloları Oluşturma

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

1 exp → term exp'
2 exp' → addop term exp'
3 exp' → λ
4 addop → +
5 addop → –
6 term → factor term'
7 term' → mulop factor term'
8 term' → λ
9 mulop → *
10 factor → (exp)
11 factor → num

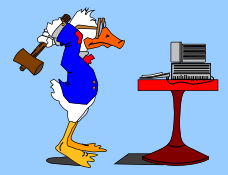
	(	)	+	-	*	num	$
exp	1					1	
exp'		3	2	2			3
addop			4	5			
term	6					6	
term'		8	8	8	7		8
mulop					9		
factor	10					11	

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) Ayrıştırma Tablosu

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



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

First(exp) = { (, num }
First(term) = { (, num }
First(factor) = { (, num }
First(addop) = { +, - }
First(mulop) = { * }

LL(1) Olmayan Gramer Sorunları

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

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 kuralsa, o zaman

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

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

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

Sadece Reduce Örneği

int * int | + int	reduce T → int
int * T | + int	reduce T → int * T
T + int |	reduce T → int
T + T |	reduce E → T
T + E |	reduce E → T + E
E |	

Shift-Reduce Ayrıştırma Örneği

|int * int + int	shift
int | * int + int	shift
int * | int + int	shift
int * int | + int	reduce $T \rightarrow$ int
int * T | + int	reduce $T \rightarrow$ int * T
T | + int	shift
T + | int	shift
T + int |	reduce $T \rightarrow$ int
T + T |	reduce $E \rightarrow T$
T + E |	reduce $E \rightarrow T + E$
E |	

Örnek 1: Shift-Reduce Ayrıştırma (1)

|int * int + int

int * int + int

↑

Örnek: Shift-Reduce Ayrıştırma (2)

|int * int + int

int | * int + int

int * int + int

↑

Örnek: Shift-Reduce Ayrıştırma (3)

|int * int + int

int | * int + int

int * | int + int

int * int + int

↑

Örnek: Shift-Reduce Ayrıştırma (4)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * int + int

↑

Örnek: Shift-Reduce Ayrıştırma (5)

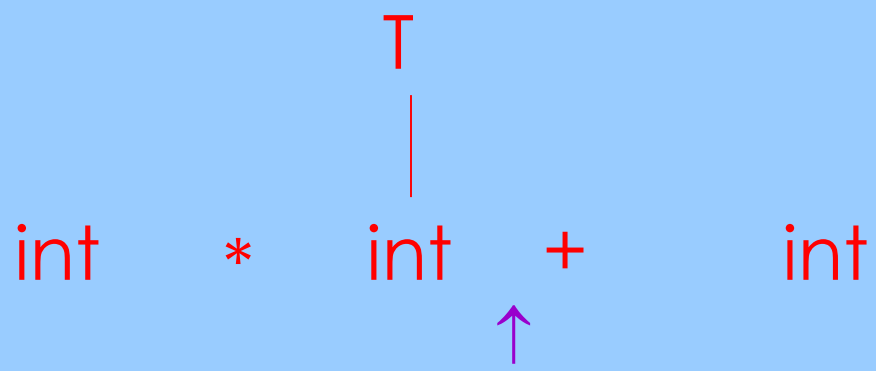
|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int



Örnek: Shift-Reduce Ayrıştırma (6)

|int * int + int

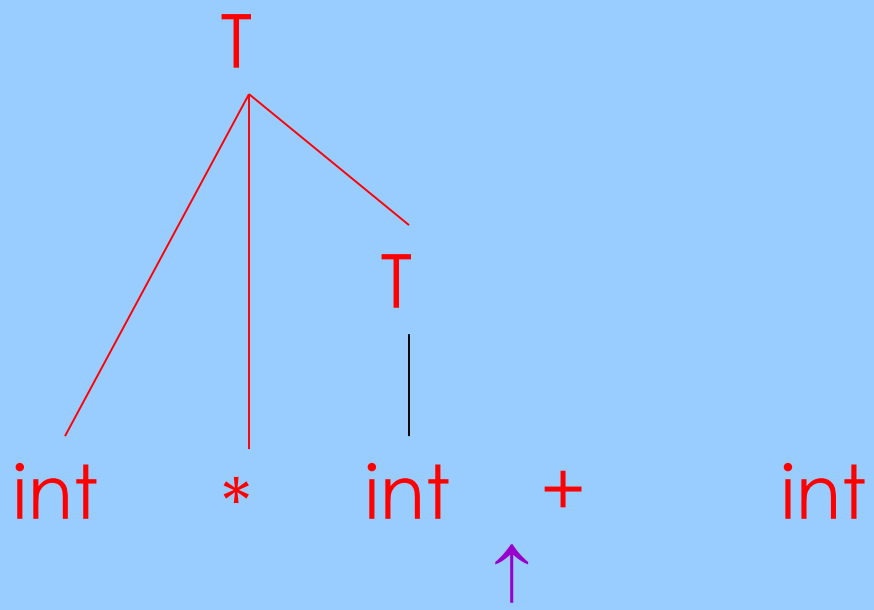
int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int



Örnek: Shift-Reduce Ayrıştırma (7)

|int * int + int

int | * int + int

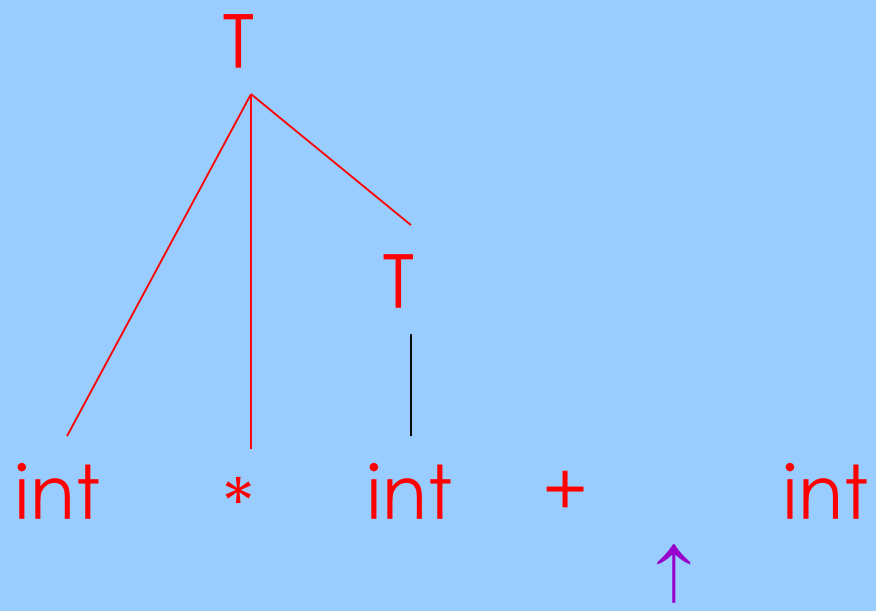
int * | int + int

int * int | + int

int * T | + int

T | + int

T + | int



Örnek: Shift-Reduce Ayrıştırma (8)

|int * int + int

int | * int + int

int * | int + int

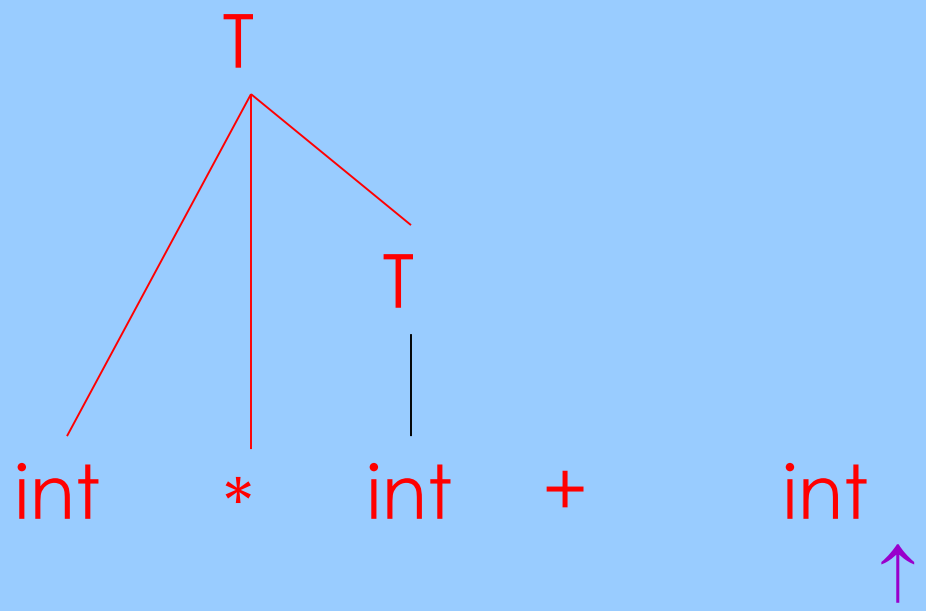
int * int | + int

int * T | + int

T | + int

T + | int

T + int |



Örnek: Shift-Reduce Ayrıştırma (9)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

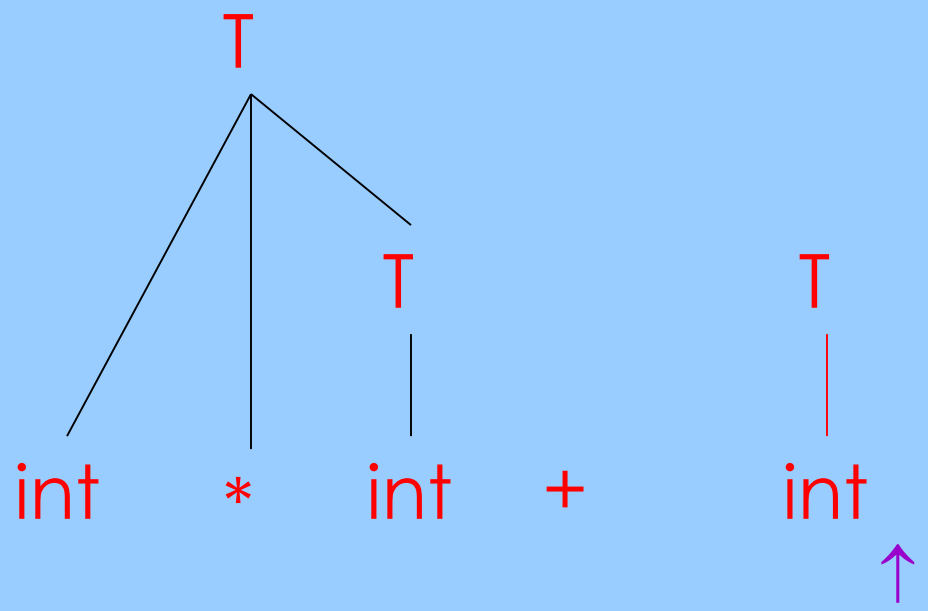
int * T | + int

T | + int

T + | int

T + int |

T + T |



Örnek: Shift-Reduce Ayrıştırma (10)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

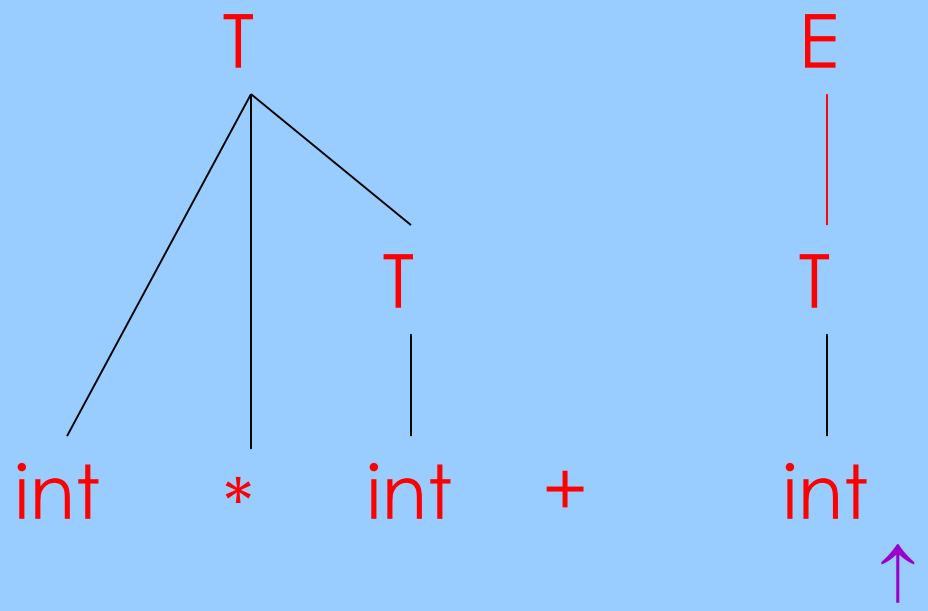
T | + int

T + | int

T + int |

T + T |

T + E |



Örnek: Shift-Reduce Ayrıştırma (11)

|int * int + int

int | * int + int

int * | int + int

int * int | + int

int * T | + int

T | + int

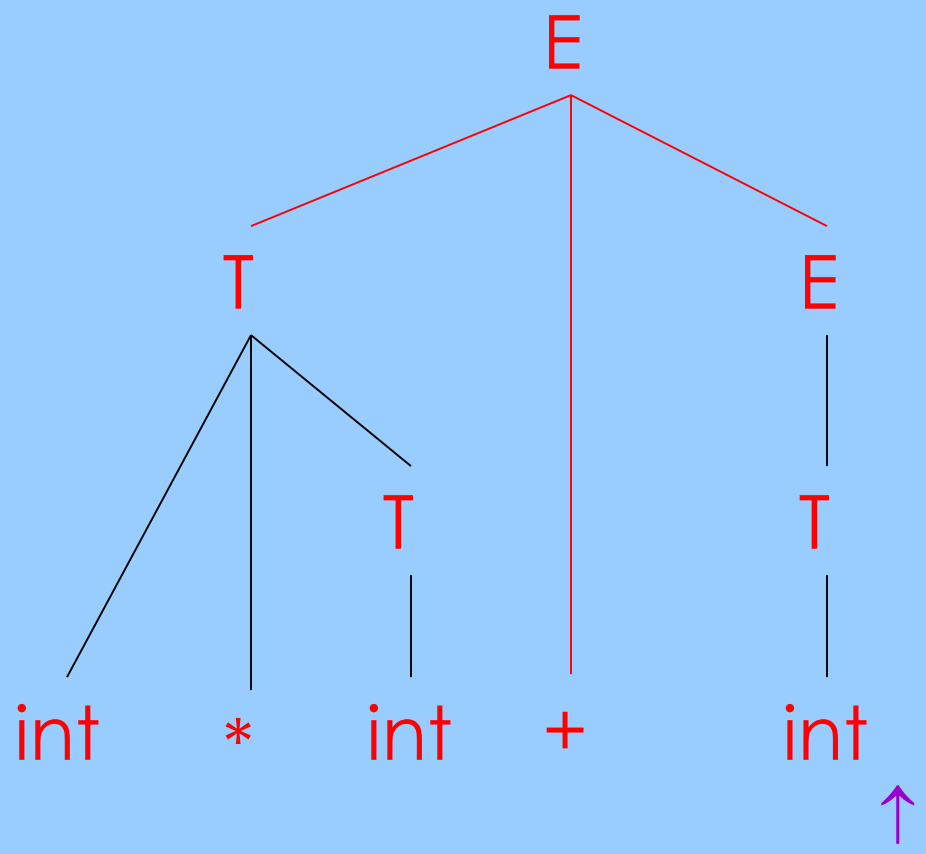
T + | int

T + int |

T + T |

T + E |

E |



Örnek 2: Shift-Reduce Ayrıştırma (1)

↑ int + (int) + (int)$ shift

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

int + (int) + (int)

Örnek: Shift-Reduce Ayrıştırma (2)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

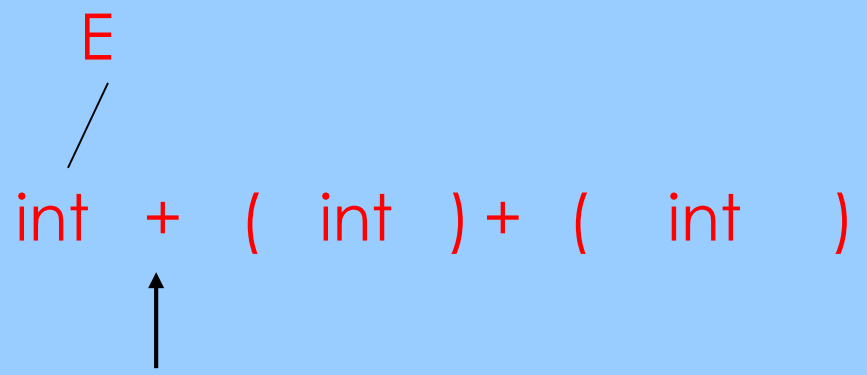
$$E \rightarrow \text{int}$$
$$E \rightarrow E + (E)$$

int + (int) + (int)

Örnek: Shift-Reduce Ayrıştırma (3)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez

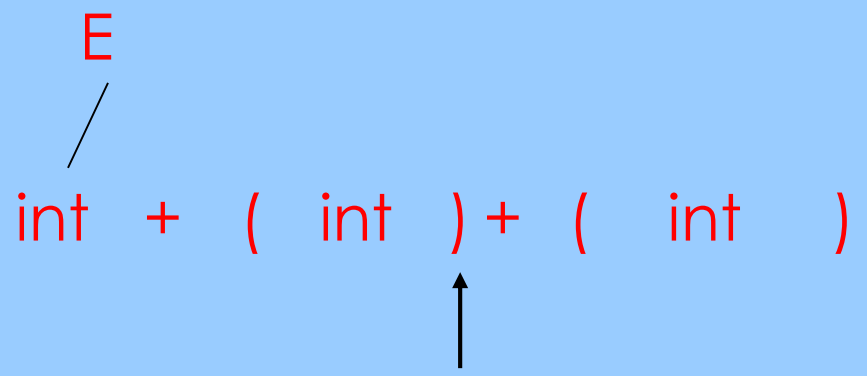
$$E \rightarrow int$$
$$E \rightarrow E + (E)$$



Örnek: Shift-Reduce Ayrıştırma (4)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int

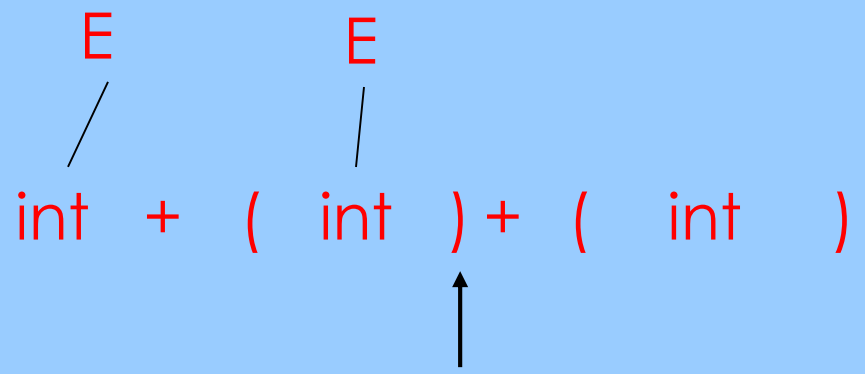
$$E \rightarrow \text{int}$$
$$E \rightarrow E + (E)$$



Örnek: Shift-Reduce Ayrıştırma (5)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int
E + (E ↑) + (int)$	shift

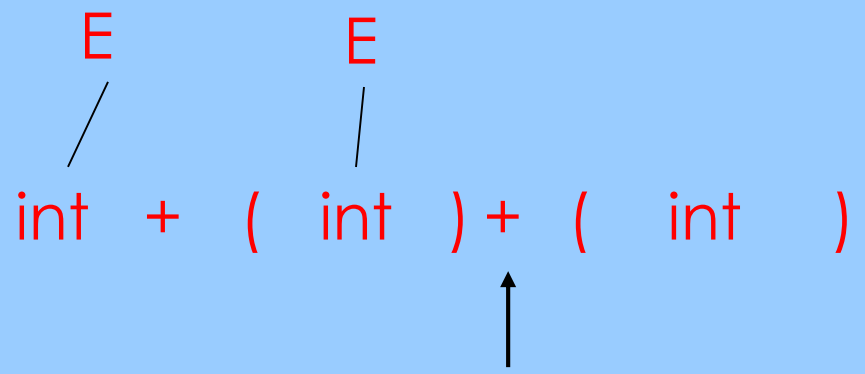
$$E \rightarrow \text{int}$$
$$E \rightarrow E + (E)$$



Örnek: Shift-Reduce Ayrıştırma (6)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. E → E + (E)

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

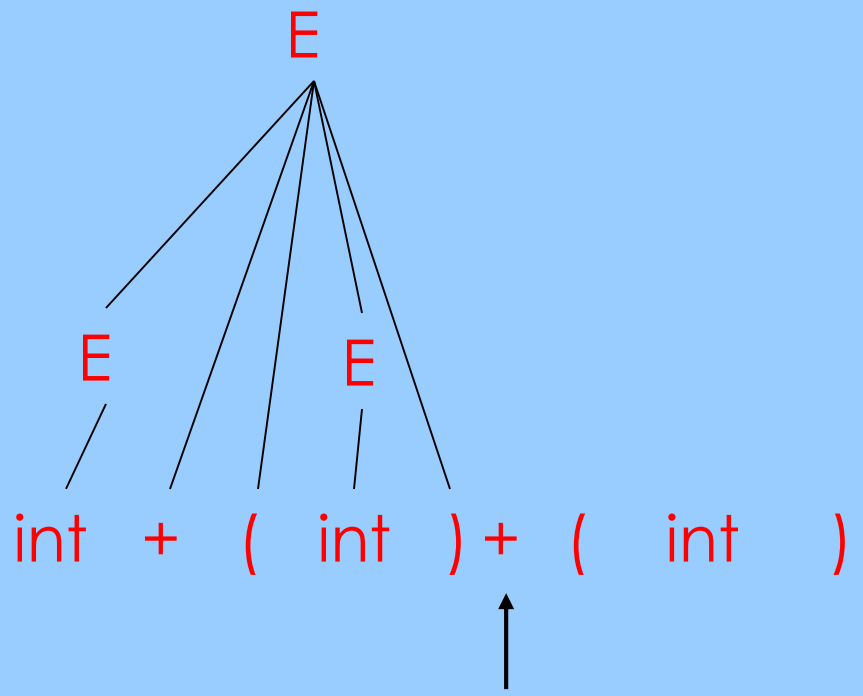


Örnek: Shift-Reduce Ayrıştırma (7)

$E \rightarrow int$

$E \rightarrow E + (E)$

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

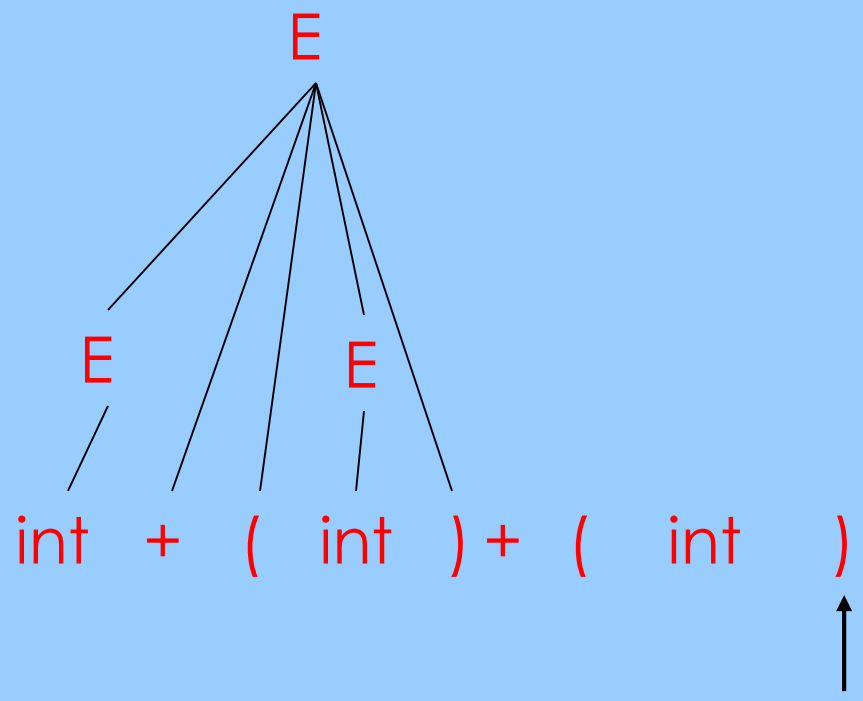


Örnek: Shift-Reduce Ayrıştırma (8)

$E \rightarrow int$

$E \rightarrow E + (E)$

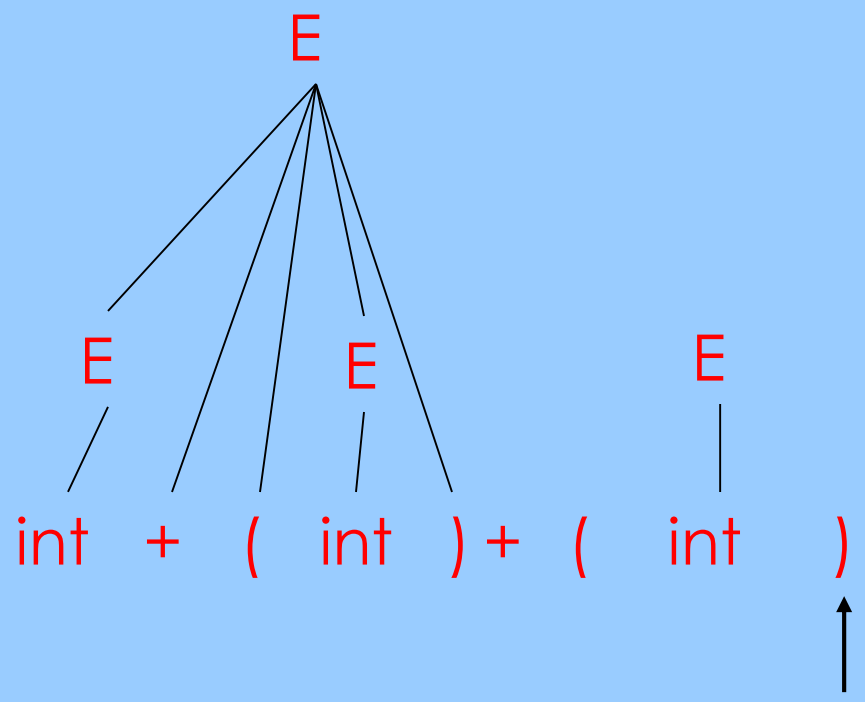
↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. $E \rightarrow int$
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. $E \rightarrow int$
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. $E \rightarrow E + (E)$
E ↑ + (int)$	shift 3 kez
E + (int ↑)$	red. $E \rightarrow int$



Örnek: Shift-Reduce Ayrıştırma (9)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. E → E + (E)
E ↑ + (int)$	shift 3 kez
E + (int ↑)$	red. E → int
E + (E ↑)$	shift

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

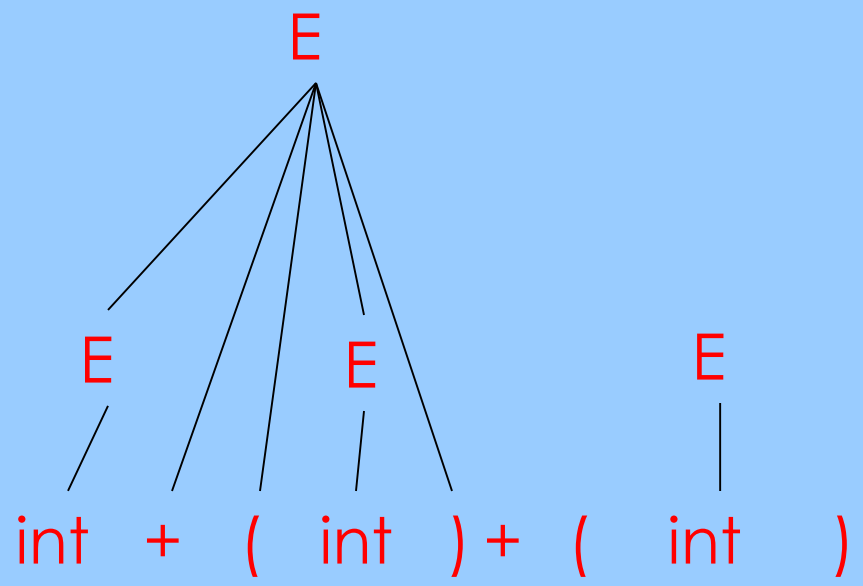


Örnek: Shift-Reduce Ayrıştırma (10)

$E \rightarrow int$

$E \rightarrow E + (E)$

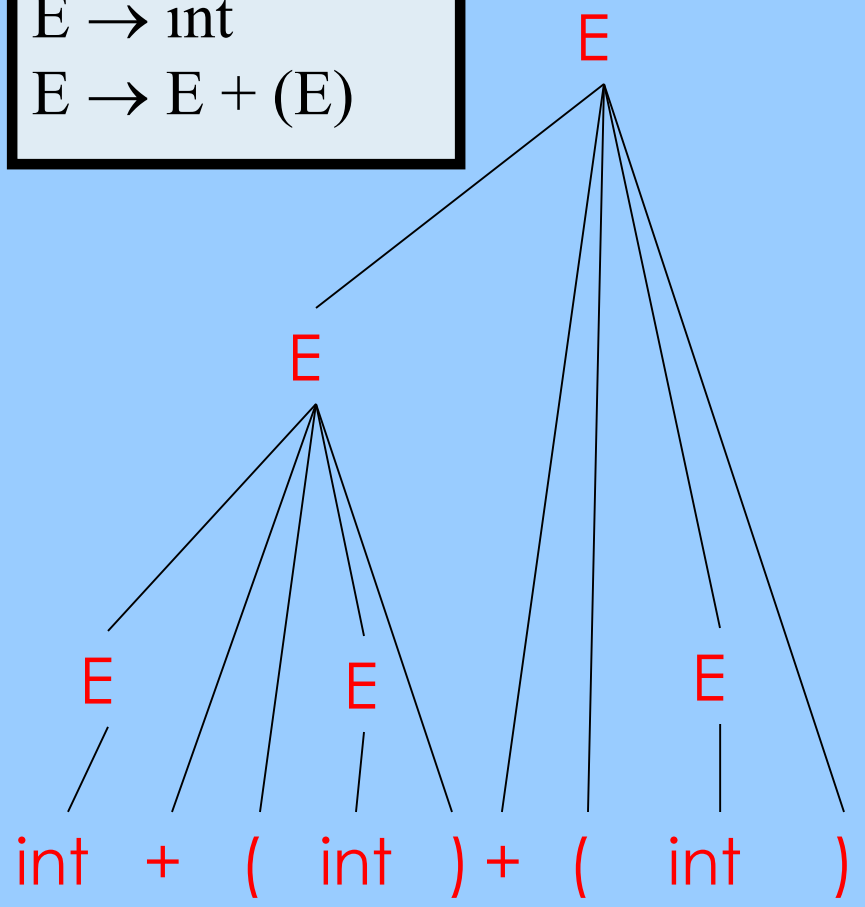
↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. $E \rightarrow int$
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. $E \rightarrow int$
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. $E \rightarrow E + (E)$
E ↑ + (int)$	shift 3 kez
E + (int ↑)$	red. $E \rightarrow int$
E + (E ↑)$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



Örnek: Shift-Reduce Ayrıştırma (11)

↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. E → E + (E)
E ↑ + (int)$	shift 3 kez
E + (int ↑)$	red. E → int
E + (E ↑)$	shift
E + (E) ↑ $	red. E → E + (E)
E ↑ $	accept

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



Örnek 3: Shift-Reduce Ayrıştırma 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 => b
$ a A	b c d e $	Shift
$ a A b	c d e $	Shift
$ a A b c	d e $	Reduce A => A b c
$ a A	d e $	Shift
$ a A d	e $	Reduce B => d
$ a A B	e $	Shift
$ a A B e	$	Reduce S => a A B
$	$	e

S => a A B e
A => A b c | b
B => 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"

Gramer:

$S \rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$A \rightarrow A\ \mathbf{b}\ \mathbf{c} \mid \mathbf{b}$

$B \rightarrow \mathbf{d}$

Bunlar productionın sağ tarafına uyar

Bir cümle indirgeme:

a **b** **b** **c** **d** **e**

a *A* **b** **c** **d** **e**

a *A* **d** **e**

a *A* *B* **e**

S

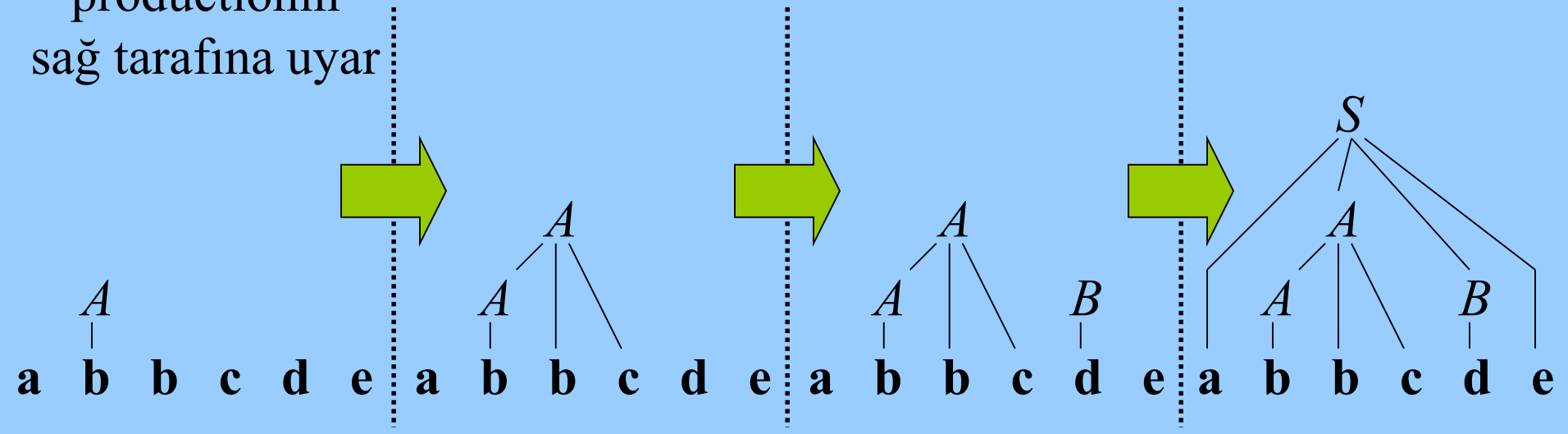
Ağaç bir sağ türetmeye uyar:

$S \Rightarrow \mathbf{a}\ A\ B\ \mathbf{e}$

$\Rightarrow \mathbf{a}\ A\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow \mathbf{a}\ A\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

$\Rightarrow \mathbf{a}\ \mathbf{b}\ \mathbf{b}\ \mathbf{c}\ \mathbf{d}\ \mathbf{e}$

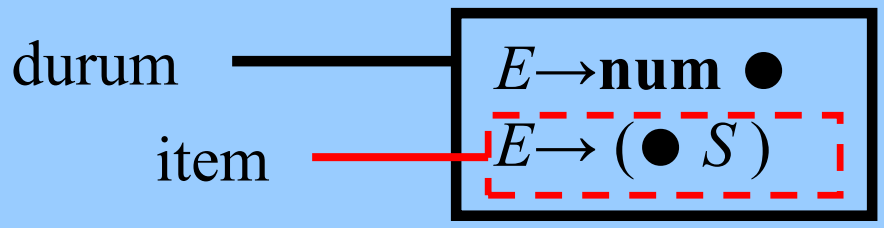


LR(0) Ayrıştırıcı

- 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 ayrıştırıcılar için bize ön hazırlık yapmaya yardım eder

LR(0) Durumları

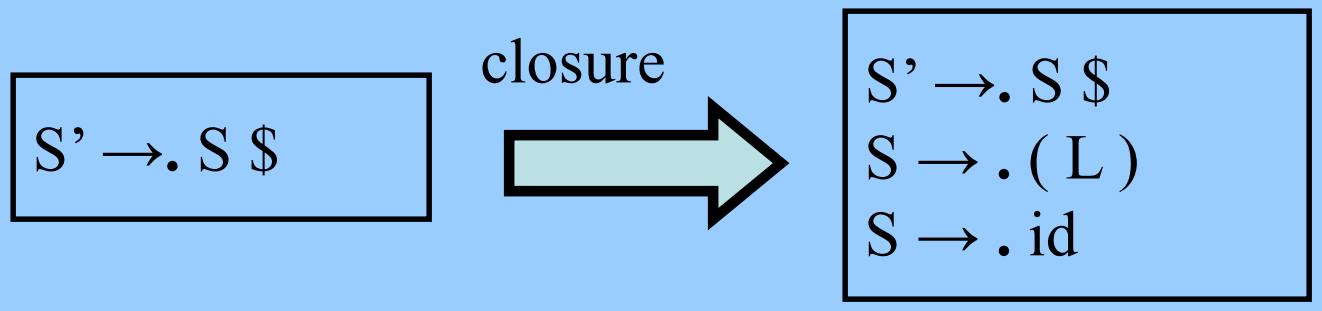
- Bir durum olası indirgemeler için ilerleme izini koruyan *item*lar kümesidir
- Bir *LR(0)* item'i i kuralın sağ tarafında bir yerde "." ayıracıyla ayrılmış dilden bir kuraldır s a production



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

Başlangıç Durumu& Kapalılık (Closure)

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

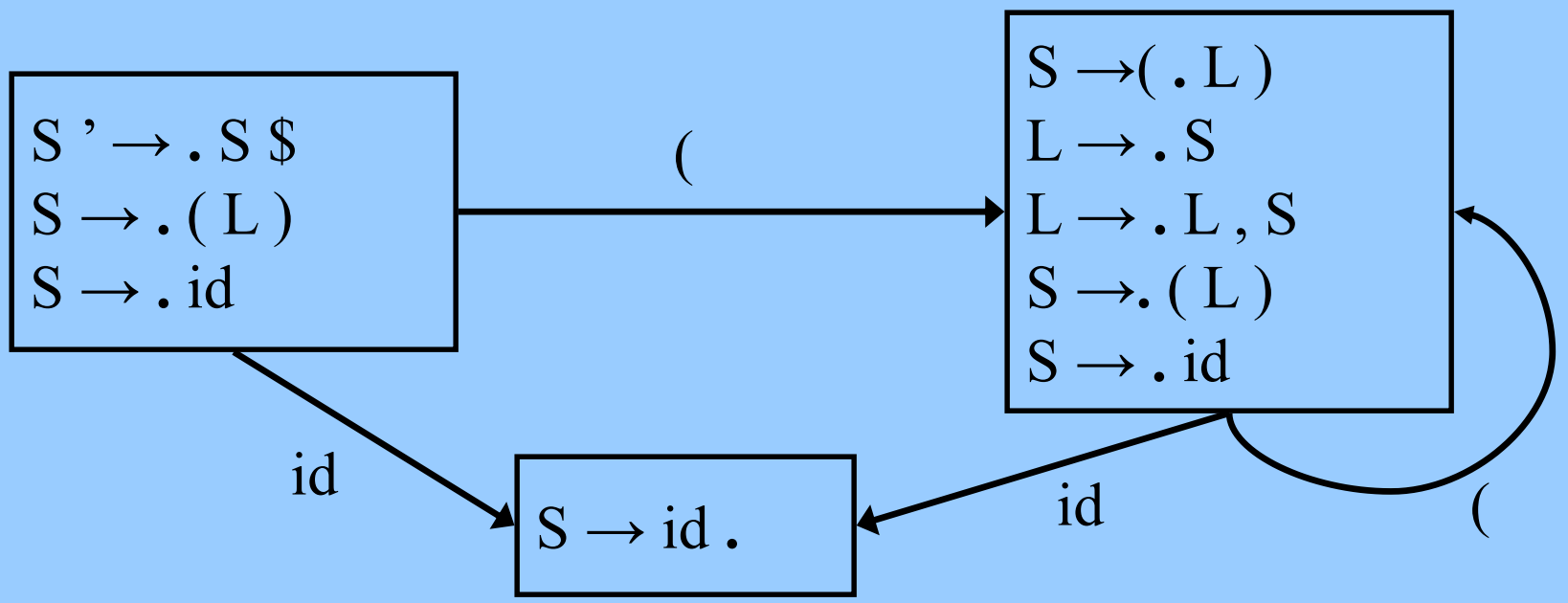


Bir yığını okuyacak DFA inşası:

- İlk adım: *S'→S $* kurallı augment gramer
- DFA'nın başlangıç durumu: boş yığın = *S'→ . 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

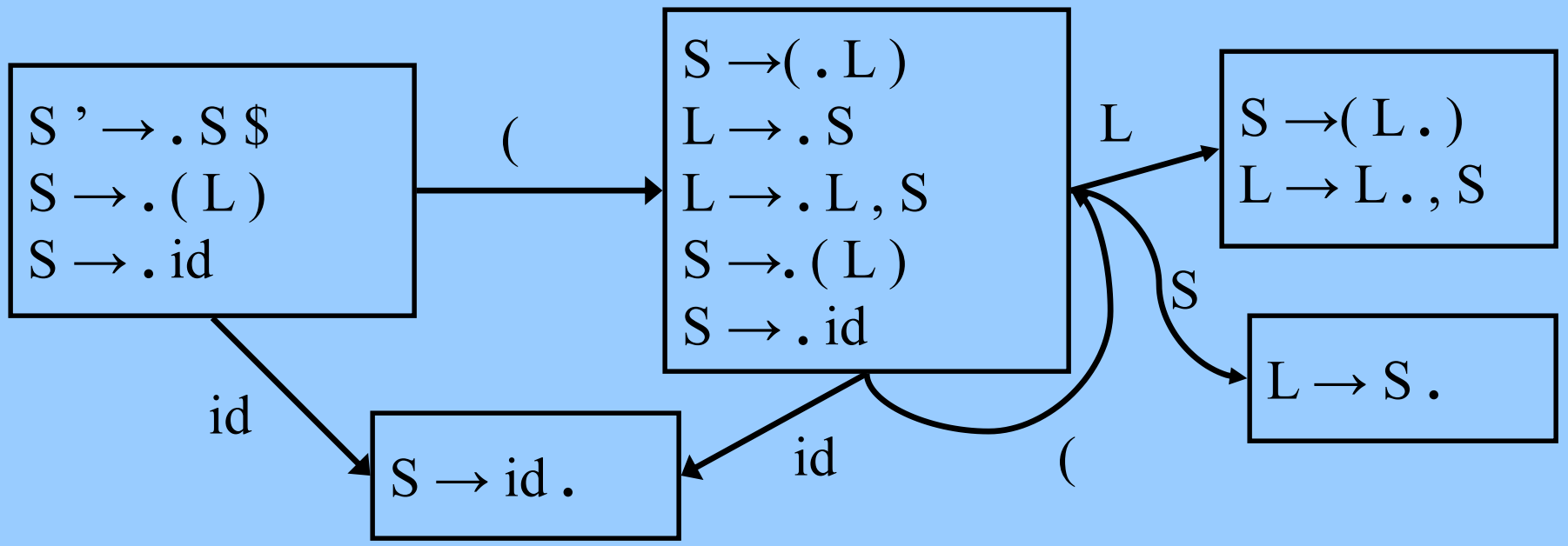
$S \rightarrow (L) \mid id$
$L \rightarrow S \mid L , S$



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

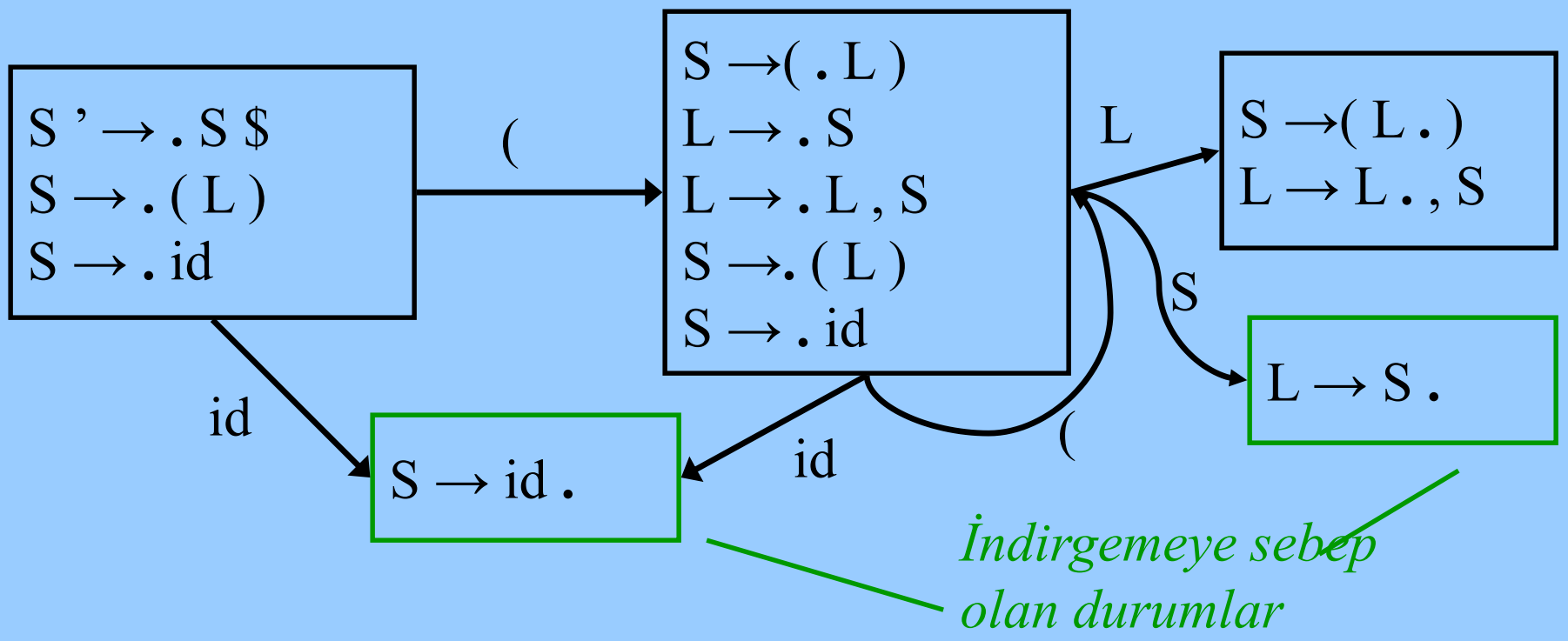
S →(L) | id
L →S | L , S



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

Reduce Hareketini Uygulama

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

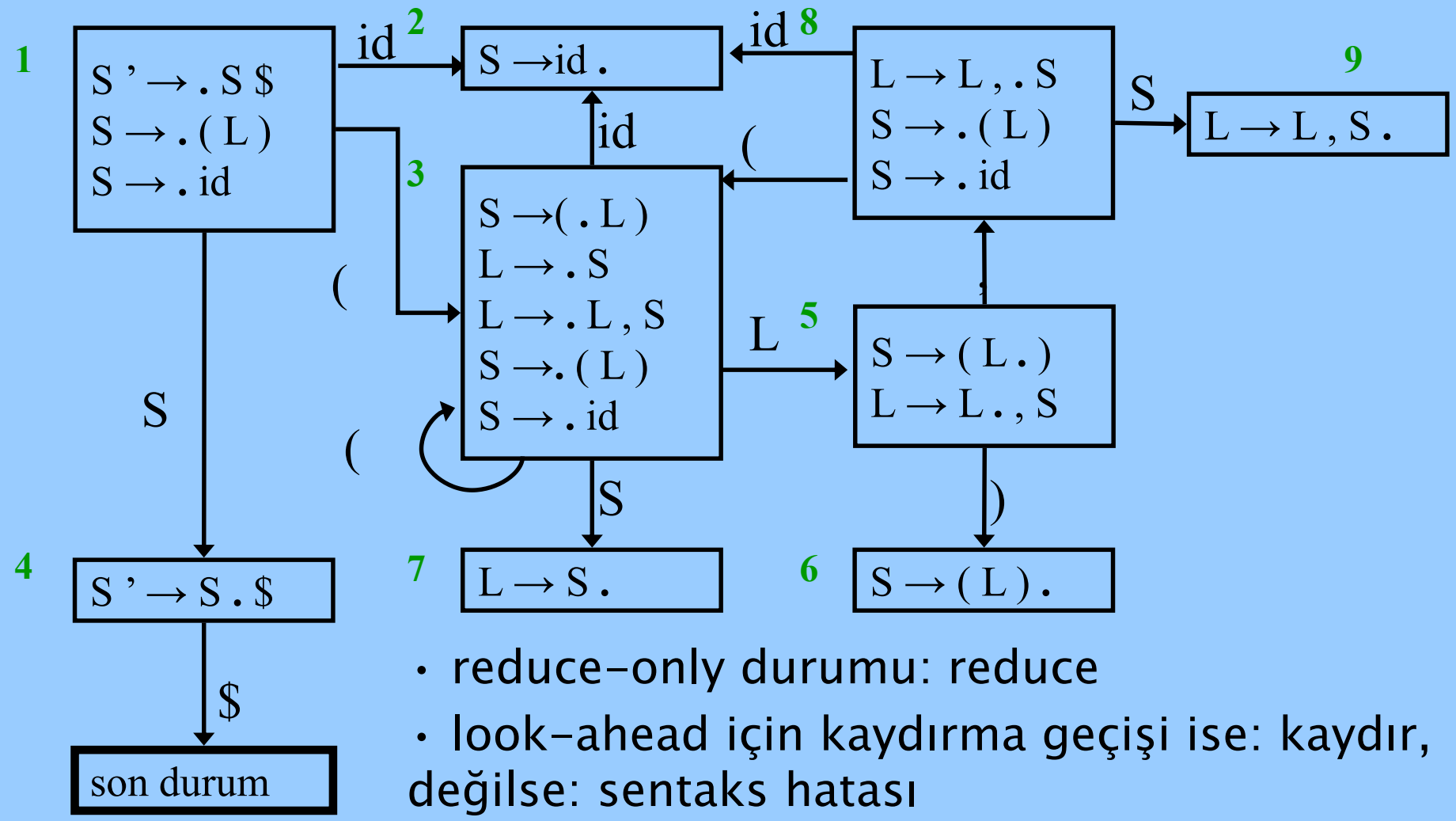


- Yığından sağ tarafı al, sol tarafla yer değiştir

X (X→γ)

Tam DFA

S →(L) | id
L →S | L , S



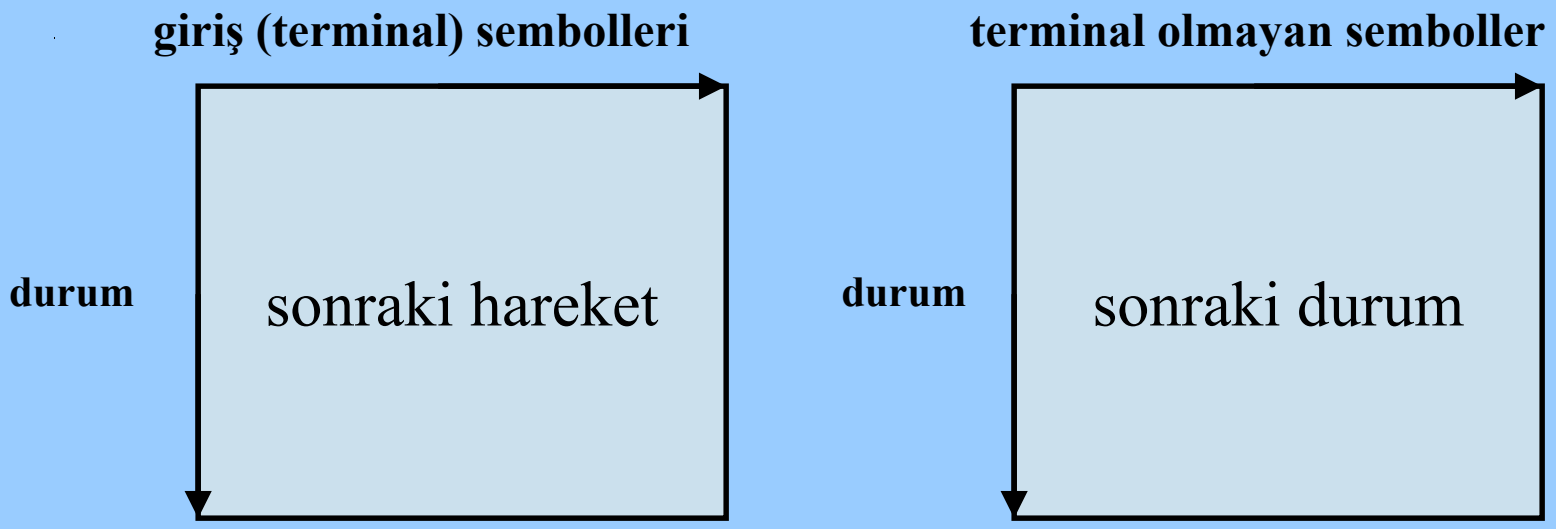
- reduce-only durumu: reduce
- look-ahead için kaydırma geçişi ise: kaydır, değilse: sentaks hatası
- mevcut durum: DFA yoluyla yığına at

Ayrıştırma Örneği: ((x),y)

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

türetme	*yığın*	*giriş*	*hareket*
((x),y) ←	1	((x),y)	shift, goto 3
((x),y) ←	1 (3	(x),y)	shift, goto 3
((x),y) ←	1 (3 (3	x),y)	shift, goto 2
((x),y) ←	1 (3 (3 x_2	),y)	reduce S→id
((*S*),y) ←	1 (3 (3 S_7	),y)	reduce L→S
((*L*),y) ←	1 (3 (3 L_5	),y)	shift, goto 6
((*L*),y) ←	1 (3 (3 L_5)$_6$	,y)	reduce S→(L)
(*S*,y) ←	1 (3 S_7	,y)	reduce L→S
(*L*,y) ←	1 (3 L_5	,y)	shift, goto 8
(*L*,y) ←	1 (3 L_5 , 8	y)	shift, goto 9
(*L*,y) ←	1 (3 L_5 , 8 y_2	)	reduce S→id
(*L*,*S*) ←	1 (3 L_5 , 8 S_9	)	reduce L→L,S
(*L*) ←	1 (3 L_5	)	shift, goto 6
(*L*) ←	1 (3 L_5)$_6$		reduce S→(L)
S	1 S_4	$	*done*

Gerçekleştirme: LR Ayrıştırma Tablosu

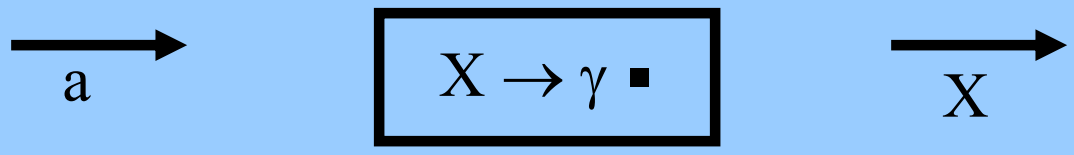


Hareket (Action) tablosu

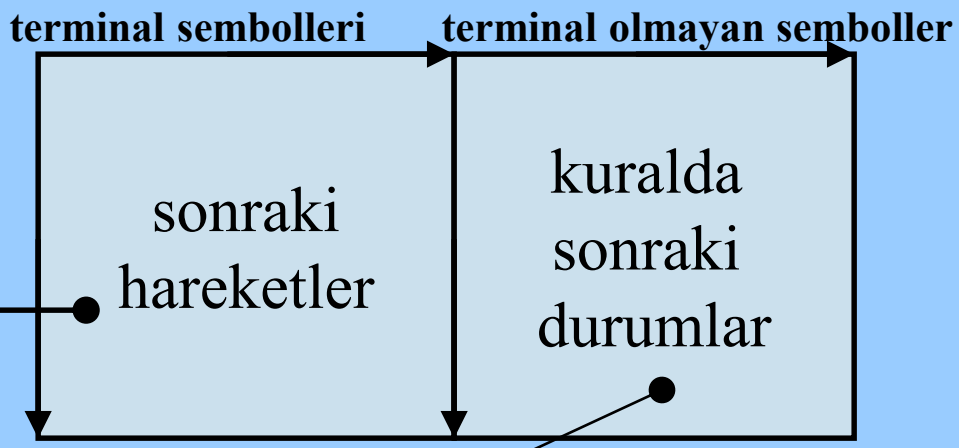
Shift ya da reduce'a karar verecek her adımda kullanılır

Git (Goto) tablosu

Bir sonraki suruma karar vermek için sadece reduce'ta kullanılır



Shift-Reduce Ayrıştırma Tablosu



Hareket tablosu

1. shift ve *n* durumuna git
2. reduce X → γ kullanarak
   - γ sembolleri yığından al
   - Yığının üstündeki durum etiketini kullanarak X in *goto tablosu*nda X'e bak ve o duruma git

- DFA + yığın = push-down automaton (PDA)

Liste Gramer Ayrıştırma 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 Ayrıştırma Örneği

- Gramer
  - $S' \rightarrow S$
  - $S \rightarrow (S)S \mid \lambda$
- Ayrıştırma hareketleri

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

- Tersi
- Sağ türetme
- Soldan sağa

1 $\Rightarrow$ **(( ))**
2 $\Rightarrow$ **(( ))**
3 $\Rightarrow$ **(( ))**
4 $\Rightarrow$ **((S))**
5 $\Rightarrow$ **((S))**
6 $\Rightarrow$ **((S) S)**
7 $\Rightarrow$ **(S)**
8 $\Rightarrow$ **(S)**
9 $\Rightarrow$ **(S) S**
10 **S'** $\Rightarrow$ **S**

Shift-reduce Ayrıştırma Örneği

- Gramer
  S' → S
  S → (S)S | λ
- Ayrıştırma hareketleri

Yığın	Giriş	Hareket
$	(( )) $	shift
$ (	( )) $	shift
$ ((	)) $	reduce S → λ
$ ((S	)) $	shift
$ ((S)	) $	reduce S → λ
$ ((S) S	) $	reduce S → (S) S
$ (S	) $	shift
$ (S)	$	reduce S → λ
$ (S) S	$	reduce S → (S) S
$ S	$	accept

Geçerli prefix

1	⇒ (( ))
2	⇒ (( ))
3	⇒ (( ))
4	⇒ ((S))
5	⇒ ((S))
6	⇒ ((S) S)
7	⇒ (S)
8	⇒ (S)
9	⇒ (S) S
10 S'	⇒ S

handle - işleyici

Shift-Reduce Ayrıştırma

S => a A B e
A => A b c | b
B => d

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 => b
$ a A	b c d e $	Shift
$ a A b	c d e $	Shift
$ a A b c	d e $	Reduce A => A b c
$ a A	d e $	Shift
$ a A d	e $	Reduce B => d
$ a A B	e $	Shift
$ a A B e	$	Reduce S => a A B
$	$	e

Örnek için Ayrıştırma 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 => a A B e

2: A => A b c

3: A => b

4: B => 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
  - Ayrıştırma yığını 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

Itemların Sonlu Otomatı (DFA)

Gramer:

S’ → S

S → (S)S

S → λ

Itemlar:

S’ → .S

S’ → S.

S → .(S)S

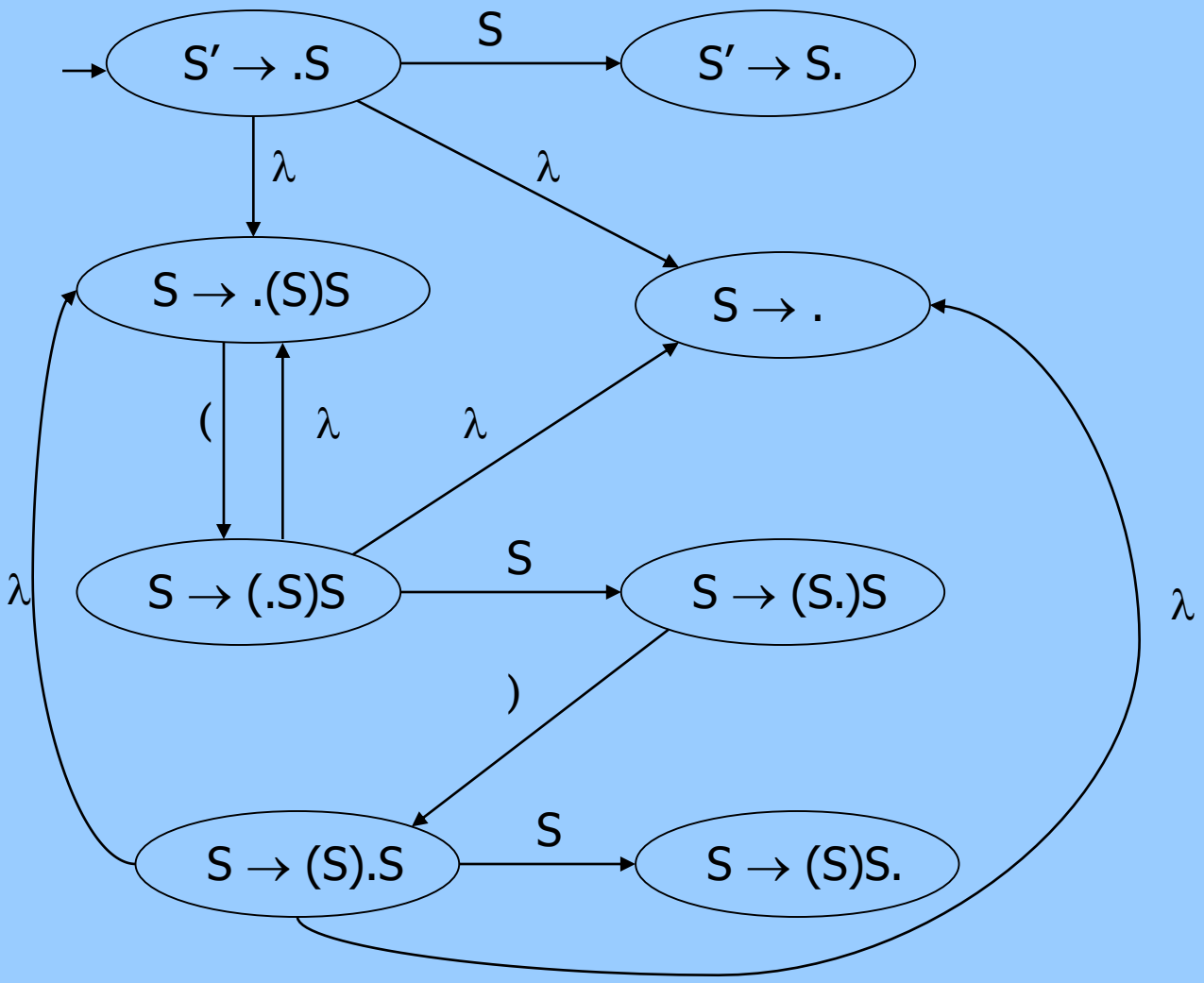
S → (.S)S

S → (S.)S

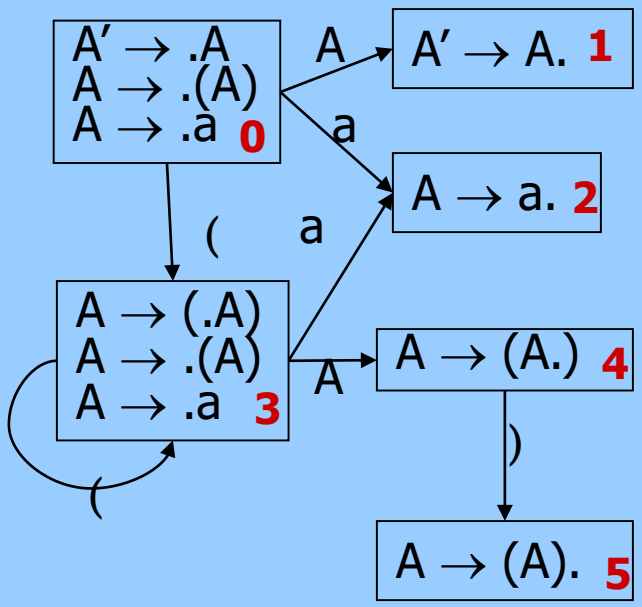
S → (S).S

S → (S)S.

S → .



LR(0) Ayrıştırma Tablosu



State	Action	Rule	(	a	)	A
0	shift		3	2		1
1	reduce	A′ -> A				
2	reduce	A -> a				
3	shift		3	2		4
4	shift				5	
5	reduce	A -> (A)				

LR(0) Ayrıştırma Örneği

Durum	Hareket	Kural	(	a	)	A
0	shift		3	2		1
1	reduce	A' -> A				
2	reduce	A -> a				
3	shift		3	2		4
4	shift				5	
5	reduce	A -> (A)				

Yığın	Giriş	Hareket
$0	((a)) $	shift
$0(3	(a)) $	shift
$0(3(3	a)) $	shift
$0(3(3a2	)) $	reduce
$0(3(3A4	)) $	shift
$0(3(3A4)5	) $	reduce
$0(3A4	) $	shift
$0(3A4)5	$	reduce
$0A1	$	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) Ayrıştırma

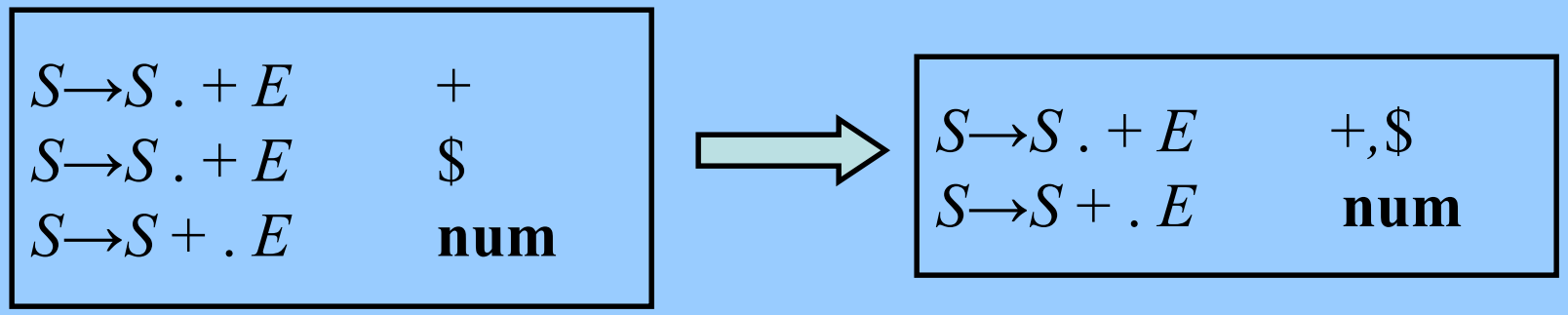
- 1 lookahead sembol ayrıştırma tablosu kadar daha güçlü
- LR(1) grameri = 1 look-ahead'li shift/reduce ayrıştırıcı tarafından tanınabilir.
- LR(1) item = LR(0) item + kuralı takip etmesi mümkün look-ahead sembolleri

LR(0): $S \rightarrow . \; S + E$

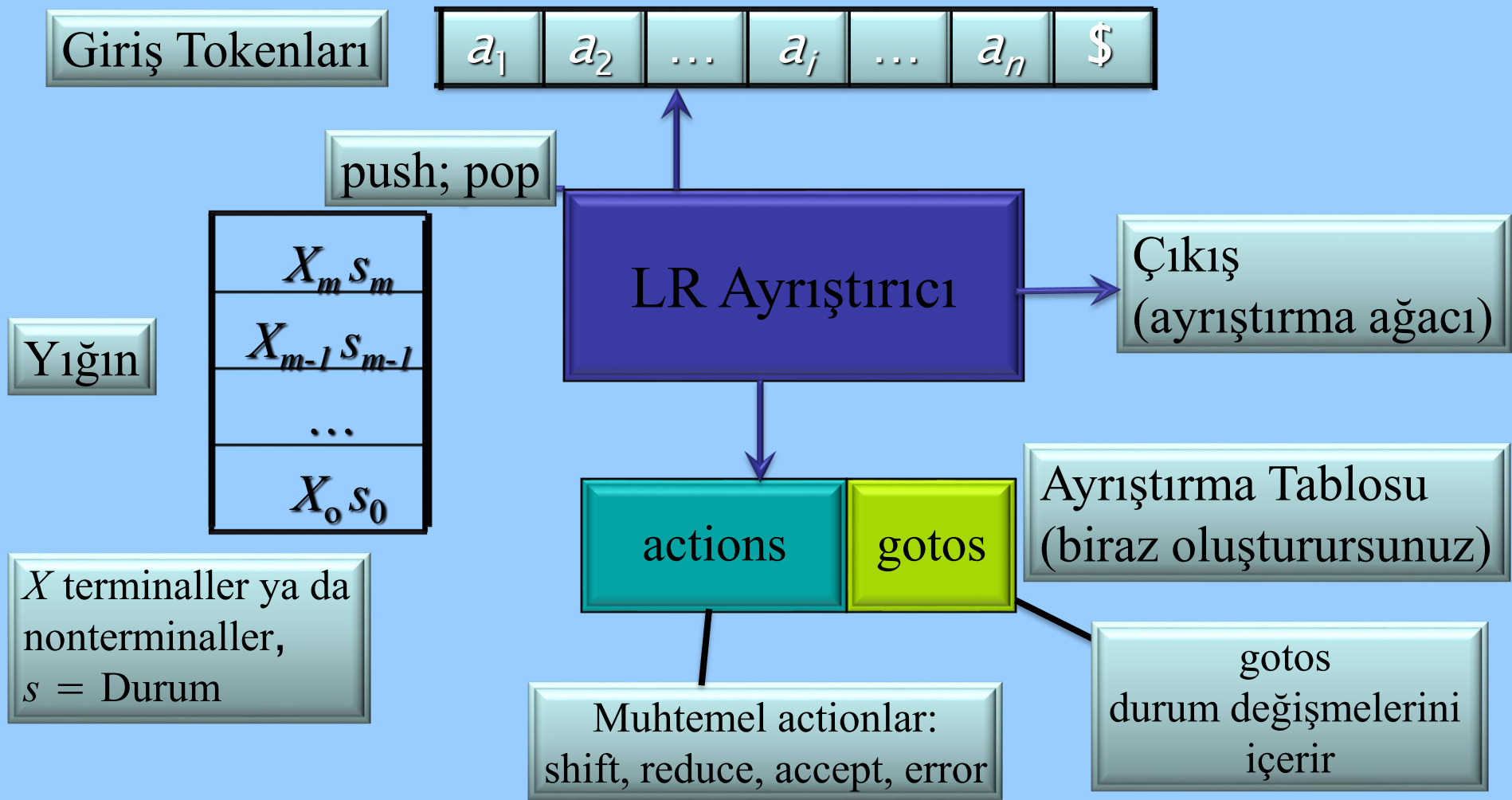
LR(1): $S \rightarrow . \; S + E \quad +$

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



Bir LR Ayrıştırıcı



4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

İşleyiciler (tanıtıcı değer) (handles) hakkında:

- Tanım: β , sağ cümlesel formun işleyicisidir (tanıtıcı değeridir) (handle)

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

$S \Rightarrow^*_{rm} \alpha A w \Rightarrow_{rm} \alpha\beta w$

- Tanım: β , sağ cümlesel formun tümceciğidir

  γ ancak ve ancak

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

- Tanım: β , sağ cümlesel form γ nın basit tümceciğidir ancak ve ancak

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

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

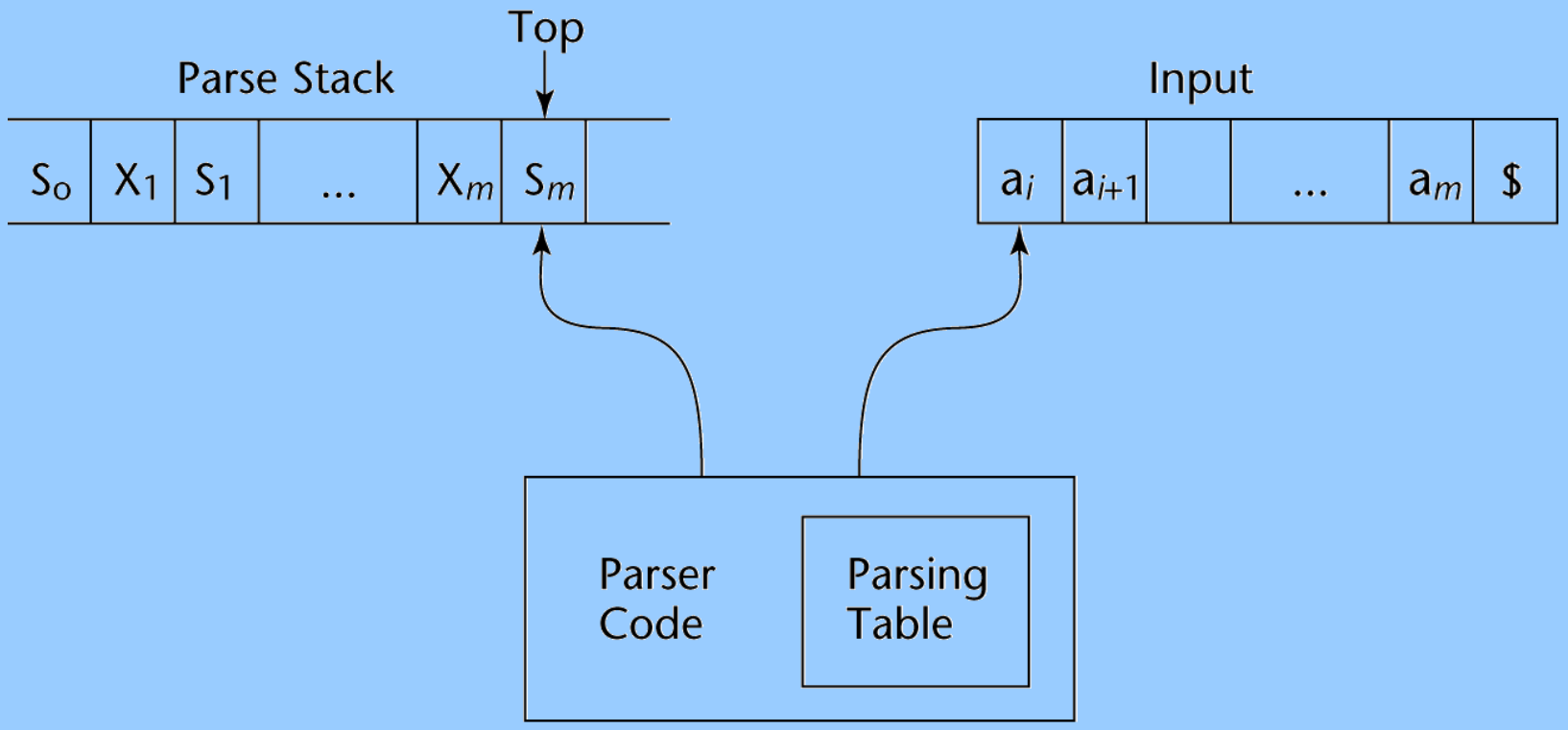
- Bir LR yapılandırması bir LR ayrıştırıcının durumunu saklar

- (S0X1S1X2S2…XmSm, aiai+1…an$)

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

Bir LR Ayrıştırıcısının (Parser) Yapısı



4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

- Başlangıç yapılandırması: $(S_0, a_1 \ldots a_n\$)$
- Ayrıştırıcı hareketleri:
  - If ACTION$[S_m, a_i]$ = Shift S, sonraki yapılandırma:

    $(S_0X_1S_1X_2S_2 \ldots X_mS_ma_iS, a_{i+1} \ldots a_n\$)$
  - If ACTION$[S_m, a_i]$ = Reduce $A \rightarrow \beta$ ve S = GOTO$[S_{m-r}, A]$, r = β nın uzunluğu olmak üzere , sonraki yapılandırma:

    $(S_0X_1S_1X_2S_2 \ldots X_{m-r}S_{m-r}AS, a_ia_{i+1} \ldots a_n\$)$

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

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

4.5 Aşağıdan-Yukarıya Ayrıştırma 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 -> E+T
| T
| E-T
T -> T*F
| F
| T/F
F -> (E)
| id
| -E
num

LR Ayrıştırma 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			

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Stack	Input	Action
0	İd+id*id$	Shift 5
0id5	+id*id$	Reduce 6 (Goto [F,0])

- Stack başlangıç durumu (0) ile başlar.
- 0 durumunda id okunursa shift 5. Shift 5'te id5 stack'e eklenir.
- 5 durumundayken girişte + var ise Reduce 6 yazılır. Reduce 6 id yerine F olduğu için Goto[F,.ç.] yazılır.
- Stack'ten id ve 5 silinir. Kalan 0 Goto[F,0]

1. $E \rightarrow E + T$
2. $E \rightarrow T$
3. $T \rightarrow T * F$
4. $T \rightarrow F$
5. $F \rightarrow (E)$
6. $F \rightarrow id$

Stack	*Input*	*Action*
0	id + id * id $	Shift 5
0id5	+ id * id $	Reduce 6 (use GOTO[0, F])
0F3	+ id * id $	Reduce 4 (use GOTO[0, T])
0T2	+ id * id $	Reduce 2 (use GOTO[0, E])
0E1	+ id * id $	Shift 6
0E1+6	id * id $	Shift 5
0E1+6id5	* id $	Reduce 6 (use GOTO[6, F])
0E1+6F3	* id $	Reduce 4 (use GOTO[6, T])
0E1+6T9	* id $	Shift 7
0E1+6T9*7	id $	Shift 5
0E1+6T9*7id5	$	Reduce 6 (use GOTO[7, F])
0E1+6T9*7F10	$	Reduce 3 (use GOTO[6, T])
0E1+6T9	$	Reduce 1 (use GOTO[0, E])
0E1	$	Accept

4.5 Aşağıdan-Yukarıya Ayrıştırma (Bottom-up Parsing) (Devam)

- Verilen bir gramerden bir araç ile bir ayrıştırıcı 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