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)

Giriş

- Sözdizimi analizörü bir derleyicinin kalbidir, çünkü anlamsal analizci ve ara kod üreteci de dahil olmak üzere birkaç önemli bileşen sözdizimi analizörünün eylemleri tarafından yönlendirilir
- Sözdizimi analizörleri doğrudan önceki bölümde işlediğimiz gramerlere dayanır
- Birinci hafta anlattığımız derleyici, yorumlayıcı ve hibrit sistemlerin üçünde de sözcüksel ve sözdizim analizcileri kullanılmaktadır.

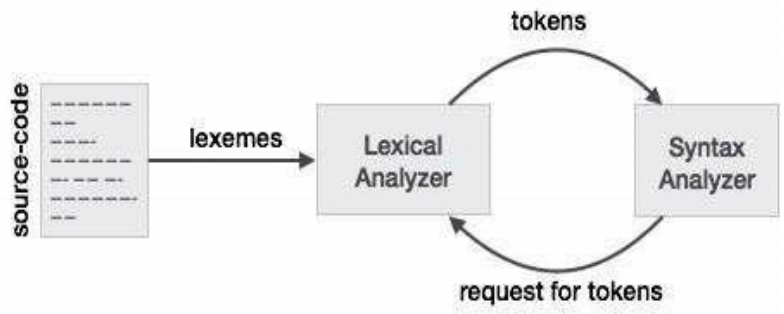
Giriş (Devamı)

- Dil(language) uygulama sistemleri, belirli uygulama yaklaşımına aldırmadan kaynak kodu (source code) analiz etmelidir. (Regular expressions)
- Hemen hemen bütün sözdizim analizörleri kaynak kodun sözdiziminin biçimsel tanımlamasına dayalıdır (BNF)

Giriş (Devamı)

- Bir dil işlemcisinin (language processor) sözdizim (syntax) analizi bölümü genellikle iki kısımdan oluşur:
- Bir düşük-düzeyli (low-level) kısım: sözcüksel analizörü (lexical analyzer), matematiksel olarak, kurallı bir gramere (regular grammar) dayalı bir sonlu otomasyon (finite automaton)
- Bir yüksek-düzeyli (high-level) kısım, sözdizim analizörü (syntax analyzer), veya ayrıştırıcı (parser)(matematiksel olarak, bağlamdan bağımsız gramere (context-free grammar) dayalı bir aşağı-itme otomasyonu (push-down automaton), veya BNF

- Sözcük analizörü isimler ve sayısal literaller gibi küçük ölçekli dil yapılarıyla ilgilenir
- Sözdizimi analizörü, statement ve bloklar gibi büyük ölçekli yapıları ele alır.



- Sözdizimini tanımlamak için BNF kullanmanın avantajı:
- Net ve özlü bir sözdizimi tanımı (syntax description) sağlar
- Ayrıştırıcı (parser) doğrudan BNF’ye dayalı olabilir
- BNF’ye dayalı ayrıştırıcıların bakımı daha kolaydır

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

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 alt stringlerini 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.

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

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
  - Sözlükler, token adı verilen sözcük kategorisiyle ilişkilendirilmiş bir karakter kalıbıyla eşleşir.
  - int value = 100;
  - Token'lar: int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).

Regular Expressions

- Sözcüksel analizci, dil kuralları tarafından tanımlanan deseni arar.
- Düzenli ifadeler, kalıpları belirlemek için önemli bir notasyondur.
- Düzenli ifadeler, özyinelemeli bir tanım örneğidir.
- Düzenli dillerin anlaşılması ve uygulanırlığı kolaydır.
  - letter = [a – z] or [A – Z]
  - digit = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 or [0-9]
  - sign = [+ | -]

Regüler İfadeler

- Düzgün İfade ..... L(R) dilinde örnek katarlar
  - rakam [0-9] .... “0”, “1”, “2”, ...
  - poztamsayı = rakam^{+} ..... “8”, “412”, ...
  - tamsayı = (-| ε) poztamsayı ..... “-23”, “34”, ...
  - reelsayı = tamsayı(ε |(.poztamsayı))“-1.56”, “12”, “1.056”
    (dikkat: bu tanım “.58” ve “45.” sözcüklerine izin vermez)
  - harf [a-z] ..... “a”, “b”, “c”,....
  - değişken_adı = harf(harf | rakam)*..... “toplam”, “sayac”,...

- Sözcüksel analizciler belirli bir girdi dizgesinden lexemeleri çıkarır ve karşılık gelen sembolleri üretir
  - int value = 100;
  - Lexeme'ler: int, value, =, 100, ;
  - Token'lar: int (keyword), value (identifier), = (operator), 100 (constant) and ; (symbol).
- Derleyicilerin ilk zamanlarında sözcüksel analizciler genelde bütün bir kaynak program dosyasını işler ve ardından token ve lexemes dosyaları üretirlerdi
- Güncel derleyicilerde bulunan sözcüksel analizciler, giriş olarak uygulanan kaynak koddaki bir sonraki lexeme'ı bulan, onun sembolünü belirleyen ve sözdizimi analizörüne geri döndüren **altprogramlardır**

- **Sözcüksel analizör genellikle bir sonraki token'a ihtiyaç duyduğunda ayrıştırıcı tarafından çağrılan bir fonksiyondur.**
- Sözcüksel analizci oluşturmaya yönelik üç yaklaşım:
  - Token'lar resmi bir tanımını yazma ve böyle bir tanımlama için tabloya dayalı (table-driven) bir sözcüksel analizci oluşturan bir yazılım aracı kullanma.
  - Token'ları tanımlayan bir durum (state) diyagramı tasarlama ve durum diyagramını uygulayan bir program yazma
  - Token'ları açıklayan bir durum diyagramı tasarlayın ve durum diyagramının tabloya dayalı bir uygulamasını manuel şekilde oluşturma
- Burada durum diyagramı üzerinde duracağız.

Durum diyagramı

- Bir durum geçiş diyagramı veya kısaca sadece durum diyagramı, yönlendirilmiş bir graftır

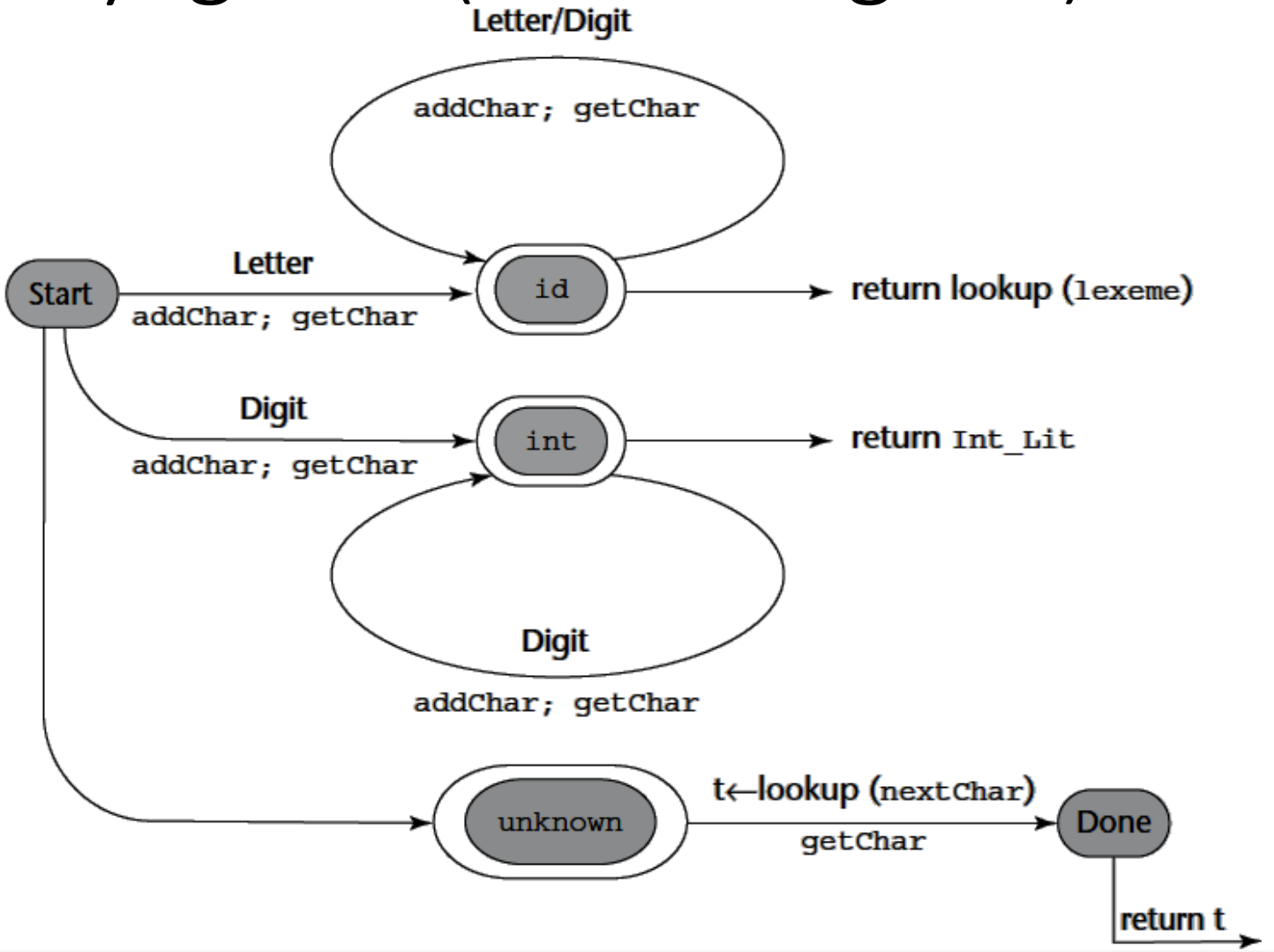
  Durum diyagramındaki düğümler, durum adlarıyla etiketlenir Oklar, hareketleri isim şeklinde içerir
- Sözcük analizciler için kullanılan formun durum diyagramları, sonlu otomata denilen matematiksel makinaların bir örneğidir
- Çoğu durumda, durum diyagramını basitleştirmek için geçişler birleştirilebilir
- Bir tanımlayıcıyı tanırken, tüm büyük ve küçük harfler eşdeğer varsayalım

  Tüm harfleri içeren bir karakter sınıfı kullanılmalı
- Bir tamsayı değişmezi tanınırken, tüm basamaklar (digits) eşdeğer varsayalım
  - Bir basamak sınıfı kullanılmalı
- Ayrılmış sözcükler ve tanımlayıcılar birlikte tanınabilir
- Olası bir tanımlayıcının aslında ayrılmış bir kelime (reverved words) olup olmadığını belirlemek için bir tablo araması kullanılmalı

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**	.					

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

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

Ayrıştırma (Parsing) Problemi (Devamı)

- Yukarıdan-aşağıya ayrıştırıcılar (Top-down parsers)
  - Bir xAα 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

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

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

- Basit deyimler (expressions) için bir gramer:

<expr> → <term> {(+ | -) <term>}

<term> → <factor> {(* | /) <factor>}

<factor> → id | (<expr>)

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.

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

```
/* Function expr
   Parses strings in the language
   generated by the rule:
   <expr> → <term> {(+ | -) <term>}
 */

void expr() {

/* Parse the first term */

  term();
/* As long as the next token is + or -, call
   lex to get the next token and parse the
   next term */

  while (nextToken == ADD_OP ||
         nextToken == SUB_OP){
    lex();
    term();
  }
}
```

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

- Birden fazla sağ tarafı (RHS) olan bir non-terminal hangi sağ tarafın ayrıştırılacağına karar vermek için bir başlangıç sürecine gereksinim duyar
  - Doğru sağ taraf girdinin bir sonraki token'ı baz alınarak (ileri bakış) seçilir
  - Bir sonraki token eşleşme bulunana kadar her sağ taraf, tarafından üretilen ilk token'la karşılaştırılır
  - Eğer eşleşme bulunmazsa bu bir sözdizimi hatasıdır

```
/* term
Parses strings in the language generated by the rule:
<term> -> <factor> {(* | /) <factor>)
*/
void term() {

/* Parse the first factor */
  factor();

/* As long as the next token is * or /,
   next token and parse the next factor */
  while (nextToken == MULT_OP || nextToken == DIV_OP) {
    lex();
    factor();
  }
} /* End of function term */
```

```
/* Function factor
   Parses strings in the language
   generated by the rule:
   <factor> -> id  |  (<expr>)  */

 void factor() {

 /* Determine which RHS */
   if (nextToken) == ID_CODE || nextToken == INT_CODE)

 /* For the RHS id, just call lex */
     lex();

/* If the RHS is (<expr>) - call lex to pass over the left parenthesis,
   call expr, and check for the right parenthesis */
  else if (nextToken == LP_CODE) {
    lex();
      expr();
     if (nextToken == RP_CODE)
        lex();
      else
        error();
    }  /* End of else if (nextToken == ...  */

   else error(); /* Neither RHS matches */
}
```

Shift-Reduce Parsing

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

- ***Shift***: |'yı bir sağa hareket ettir
  - Bir terminali sol alt stringe kaydır

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

$$E + (| int) \Rightarrow E + (int |)$$

- ***Reduce:*** Sol alt stringin sağında *ters kural (üretim)*
  - $A \rightarrow xy$ bir kuralsa, o zaman

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

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

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

Örnek : Shift-Reduce Ayrıştırma (1)

↑ int + (int) + (int)$ shift

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

int + (int) + (int)

Örnek: Shift-Reduce Ayrıştırma (2)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

E → int
E → E + (E)

int + (int) + (int)

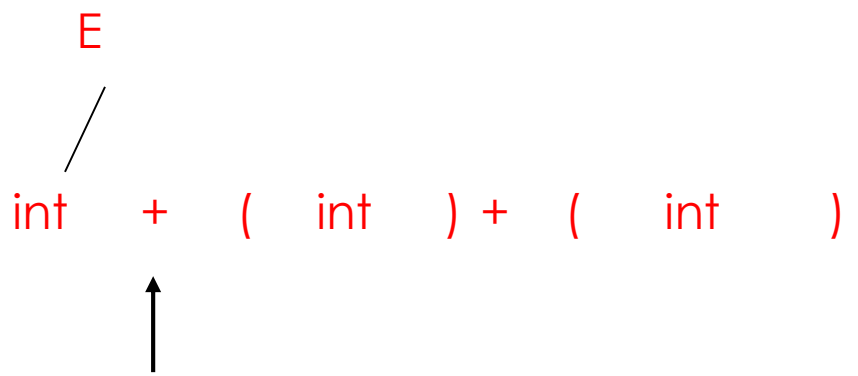
Örnek: Shift-Reduce Ayrıştırma (3)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

E ↑ + (int) + (int)$ shift 3 kez

E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (4)

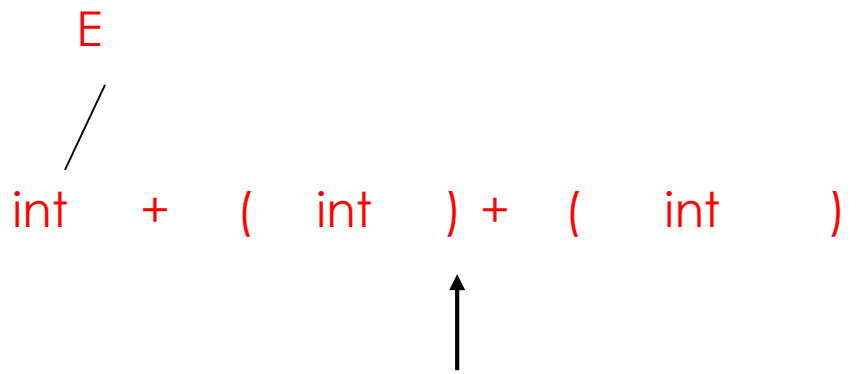
↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

E ↑ + (int) + (int)$ shift 3 kez

E + (int ↑) + (int)$ red. E → int

E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (5)

↑ int + (int) + (int)$ shift

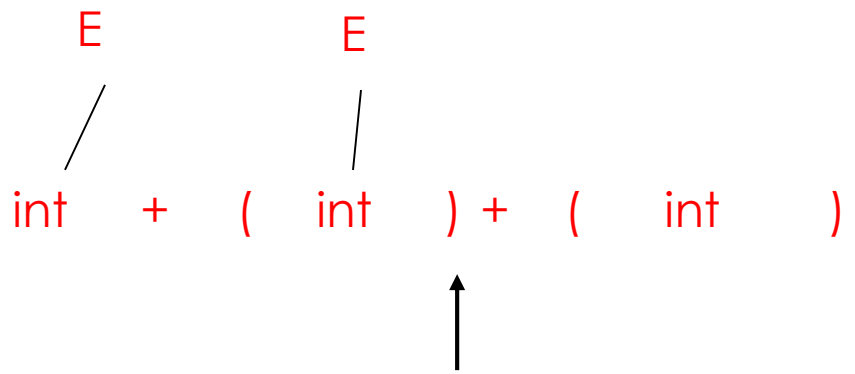
int ↑ + (int) + (int)$ red. E → int

E ↑ + (int) + (int)$ shift 3 kez

E + (int ↑) + (int)$ red. E → int

E + (E ↑) + (int)$ shift

E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (6)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

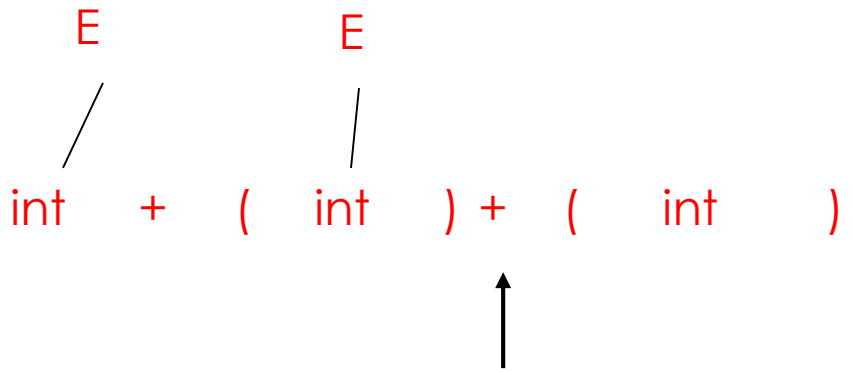
E ↑ + (int) + (int)$ shift 3 kez

E + (int ↑) + (int)$ red. E → int

E + (E ↑) + (int)$ shift

E + (E) ↑ + (int)$ red. E → E + (E)

E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (7)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

E ↑ + (int) + (int)$ shift 3 kez

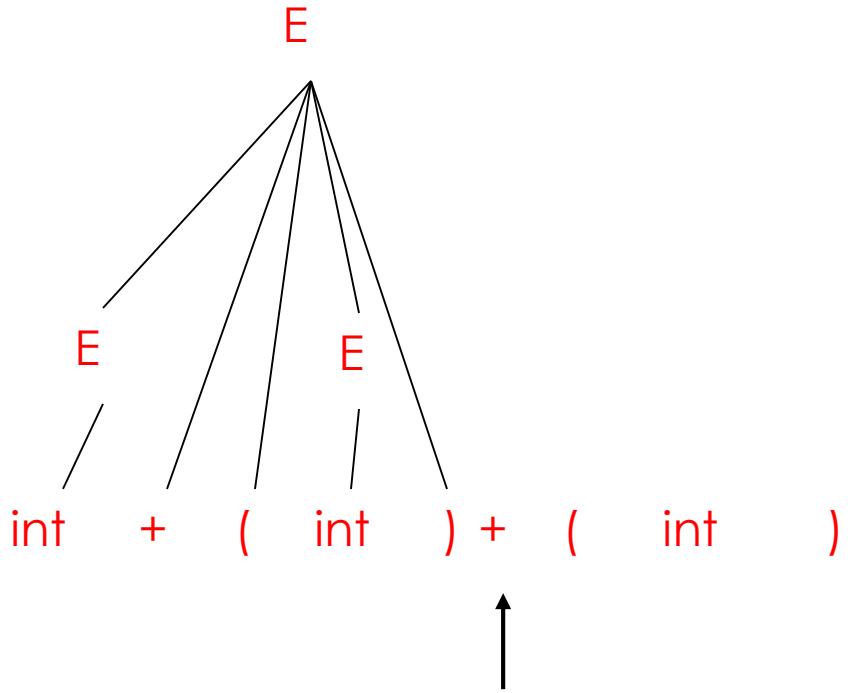
E + (int ↑) + (int)$ red. E → int

E + (E ↑) + (int)$ shift

E + (E) ↑ + (int)$ red. E → E + (E)

E ↑ + (int)$ shift 3 kez

E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (8)

↑ int + (int) + (int)$ shift

int ↑ + (int) + (int)$ red. E → int

E ↑ + (int) + (int)$ shift 3 kez

E + (int ↑) + (int)$ red. E → int

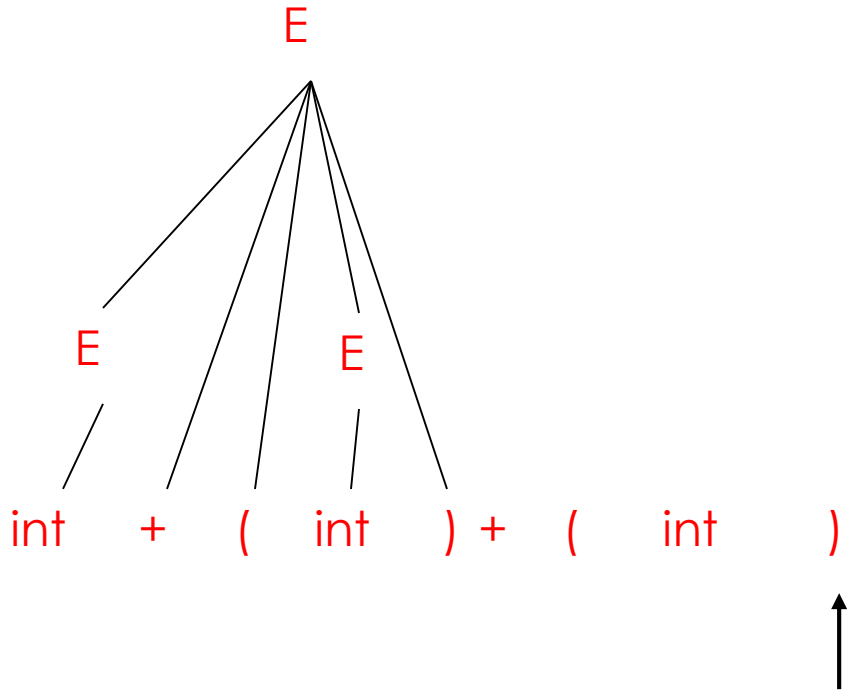
E + (E ↑) + (int)$ shift

E + (E) ↑ + (int)$ red. E → E + (E)

E ↑ + (int)$ shift 3 kez

E + (int ↑)$ red. E → int

E → int
E → E + (E)



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

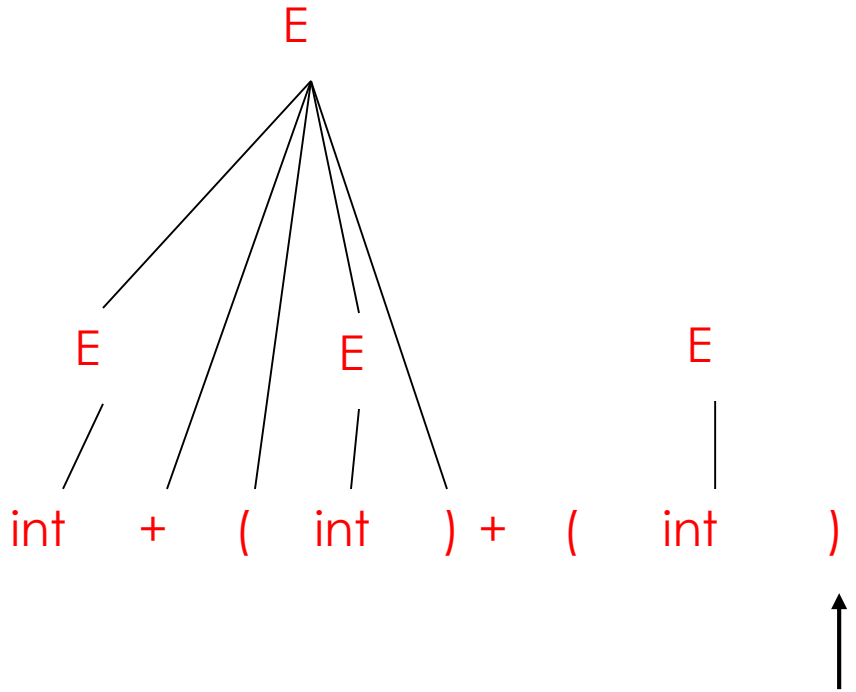
E + (E) ↑ + (int)$ red. E → E + (E)

E ↑ + (int)$ shift 3 kez

E + (int ↑)$ red. E → int

E + (E ↑)$ shift

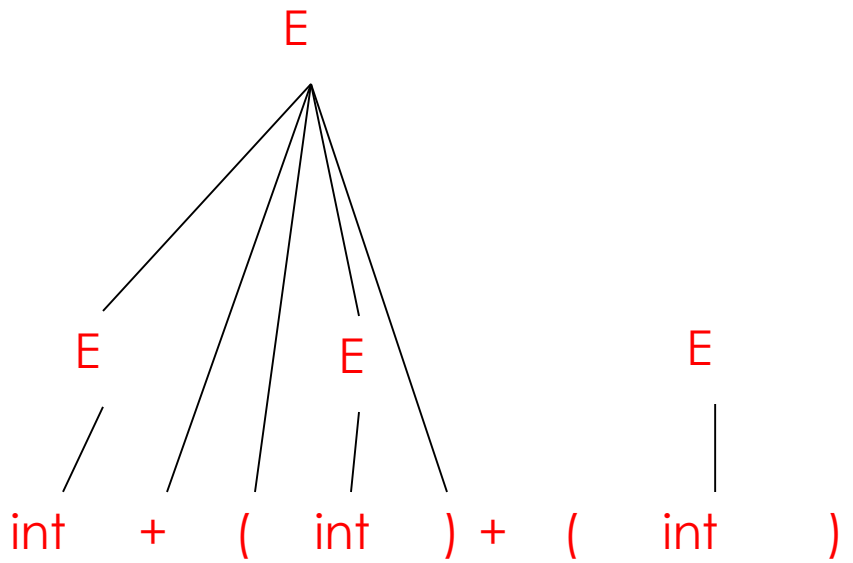
E → int
E → E + (E)



Örnek: Shift-Reduce Ayrıştırma (10)

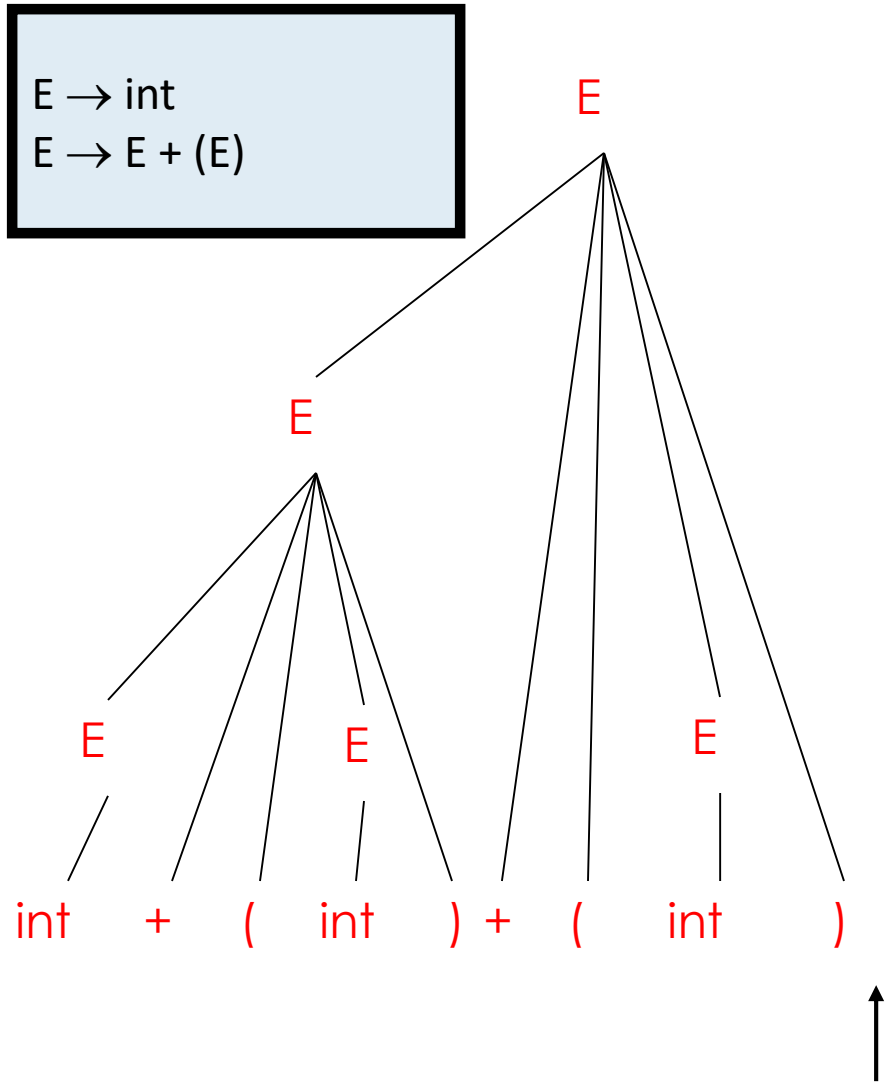
↑ int + (int) + (int)$	shift
int ↑ + (int) + (int)$	red. E → int
E ↑ + (int) + (int)$	shift 3 kez
E + (int ↑) + (int)$	red. E → int
E + (E ↑) + (int)$	shift
E + (E) ↑ + (int)$	red. E → E + (E)
E ↑ + (int)$	shift 3 kez
E + (int ↑)$	red. E → int
E + (E ↑)$	shift
E + (E) ↑ $	red. E → E + (E)

E → int
E → 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



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

Step	Parse Stack	Look Ahead	Unscanned	Parser Action
0	*empty*	*id*	= B + C*2	Shift
1	*id*	=	B + C*2	Shift
2	*id* =	*id*	+ C*2	Shift
3	*id* = *id*	+	C*2	Reduce by Value ← *id*
4	*id* = Value	+	C*2	Reduce by Products ← Value
5	*id* = Products	+	C*2	Reduce by Sums ← Products
6	*id* = Sums	+	C*2	Shift
7	*id* = Sums +	*id*	*2	Shift
8	*id* = Sums + *id*	*	2	Reduce by Value ← *id*
9	*id* = Sums + Value	*	2	Reduce by Products ← Value
10	*id* = Sums + Products	*	2	Shift
11	*id* = Sums + Products *	*int*	*eof*	Shift
12	*id* = Sums + Products * *int*	*eof*		Reduce by Value ← *int*
13	*id* = Sums + Products * Value	*eof*		Reduce by Products ← Products * Value
14	*id* = Sums + Products	*eof*		Reduce by Sums ← Sums + Products
15	*id* = Sums	*eof*		Reduce by Assign ← *id* = Sums
16	Assign	*eof*		Done

Grammar

Assign ← *id* = Sums

Sums ← Sums + Products

Sums ← Products

Products ← Products * Value

Products ← Value

Value ← *int*

Value ← *id*