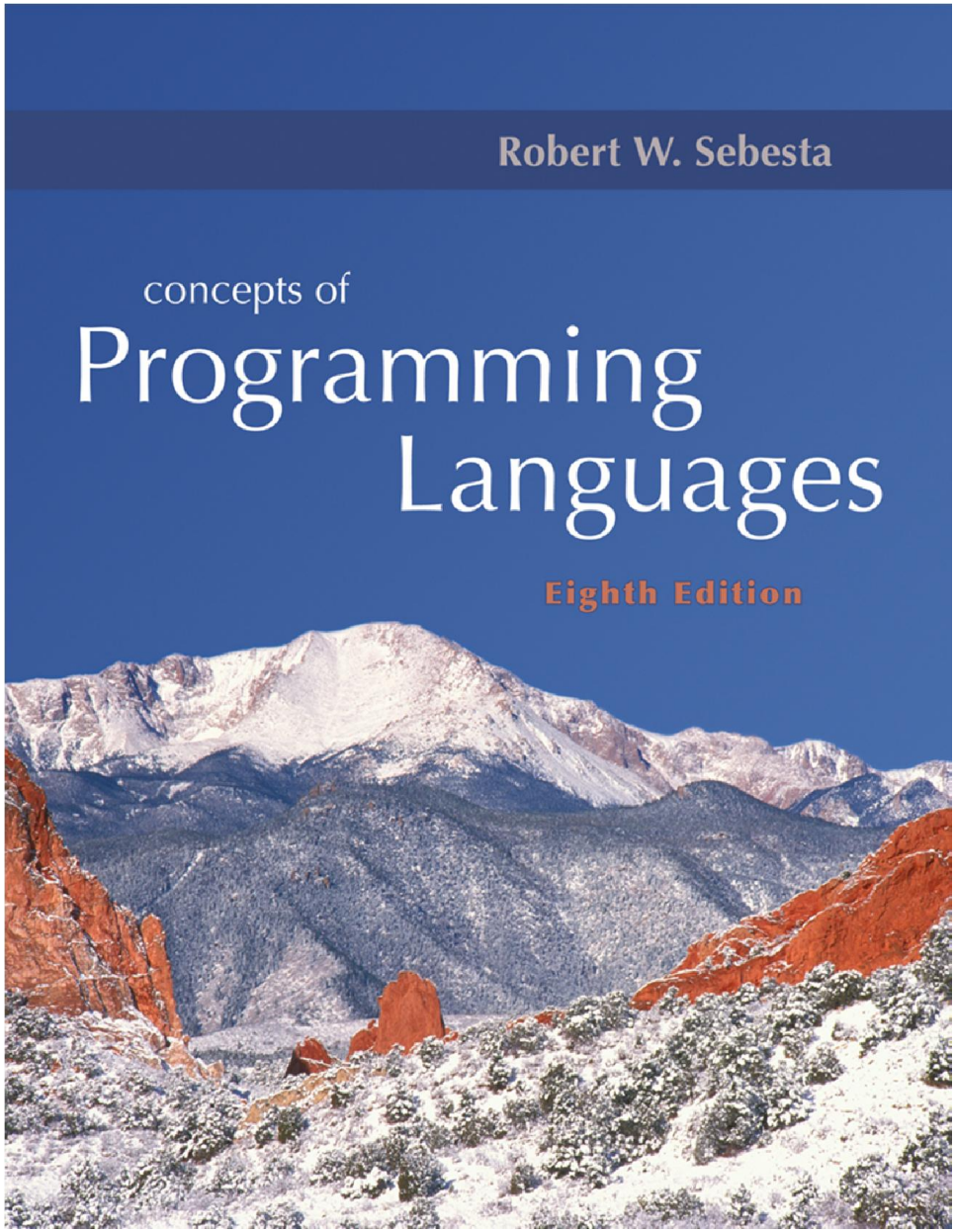
Bölüm 4

Sözcük ve Sentaks Çözümlemesi (Analizi)



ISBN 0-321-49362-1

Bölüm 4 Konuları

- Giriş
- Sözcük çözümlemesi
- Ayrıştırma (parsing) problemi
- Özyineleyerek inen ayrıştırma (Recursive-Descent Parsing)
- Aşağıdan-yukarı ayrıştırma (Bottom-Up Parsing)

Giriş

- Dilin gerçeklenme yöntemi ne olursa olsun, kaynak kodun analizi gereklidir.
- Hemem hemen tüm sentaks analiz yöntemleri dilin senataksının matematiksel tanımına bağlıdır (BNŞ)

Sentaks Analizi

- Dil işlemcisinin iki kısmı var:
  - *Sözcük analizcisi (düşük seviyeli,* matematiksel olarak *düzgün ıfadeyeden* üretilmiş bir *sonlu özdevinir*)
  - *Ayrıştırıcı* (*yüksek seviyeli,* matematiksel olarak *ortam bağımsız gramerden* üretilmiş *yığınlı özdevinir*)

Sentaks tarifi için BNŞ kullanmanın avantajları

- Kısa ve öz sentaks tanımı sağlar
- Ayrıştırıcı direkt olarak BNŞden elde edilebilir
- BNŞden elde edilen ayrıştırıcıların bakımı kolaydır

Niye sözcük analizci ve ayrıştırıcı ayrı olarak var?

- *Sadelik* – sözcük analizcisi ayrıştıtıcıya göre daha basit; onu ayırmak ayrıştırıcıyı da daha basit hale getirir
- *Verimlilik* – ayırmak sözcük analizcisinin optimizasyonuna (eniyileme) fırsat tanır
- *Taşınabilirlik*– sözcük analizcisinin bazı kısımları taşınır olmayabilir, ama ayrıştırıcı her zaman taşınabilirdir.

Sözcük Analizi

- Sözcük analizcisi karakter dizisi için desen eşleştiricidir
- Ayrıştırıcının "ön yüzü" dür
- Kaynak programın bır arada anlamlı karakter dizilerini (*sözcükleri)* belirler
  - Sözcükler *jeton* denen dilin kelimelerinin kategorileri ile ilişkili karakter desenlerine uyarlar
  - "ogrenciYasi" ve "ortalama" birer sözcüktür; her ikisinin de jetonu DEGISKEN_ADI olabilir.

Sözcük Analizi ...

- Sözcük analizcisi genellikle ayrıştırıcının bir sonraki jetonu göndermesi için gerektikçe çağırdığı bir fonksiyondur.
- Sözcük analizcisi yapmak için üç yaklaşım:
  i. Sözcükleri "düzenli deyim"lerle tanımla, jetonlarla ilişkilendir. LEX (FLEX) gibi bir yazılıma ver. LEX (FLEX) önce "düzenli deyim"leri durum geçiş tablosuna çevirir, sonra bu tabloyu kullanan bir "sözcük analizcisi" fonksiyonu üretir. Biz o fonksiyonu ayrıştırıcımızda kullanırız.
  ii. Sözcükleri tanıyan bir "durum geçiş diyagram"ı tasarla, bu diyagramı gerçekleştiren (onun yaptığını yapan) bir program yaz (kullanacağımız yaklaşım)

Sözcük Analizi ...

iii. Sözcükleri tanıyan bir “durum geçiş diyagram”ı tasarla, bu diyagramı tablo halinde bilgisayarda temsil et, sonra tabloyu kullanan bir fonksiyon yaz. Bu fonksiyon genel amaçlı olur, çünkü tablonun içeriğinden bağımsız olarak tabloyu “çalıştırır”. Bu yöntemde LEX’in yaptığını manüel olarak yapıyoruz. “Düzenli deyim” aşamasından geçmeden direkt tabloyu tasarlıyoruz. Ama daha zor bir yaklaşım.

Geçiş diyagramı tasarımı

- Safça tasarlanmış bir geçiş diyagramında her harf için bir geçiş olurdu. Diyagram çok büyük olurdu!

Geçiş diyagramı tasarımı ...

- Birçok durumda diyagramı sadeleştirmek için geçişleri birleştirebiliriz
  - İsim (identifier) tanırken, karakter sınıfı kullan ([a..zA..Z] = {a,b,c,….,z,A,B,…,Z})
  - Sayı tanırken, rakam sınıfı kullan ([0..9] = {0,1,2,…, 9})

Geçiş diyagramı tasarımı ...

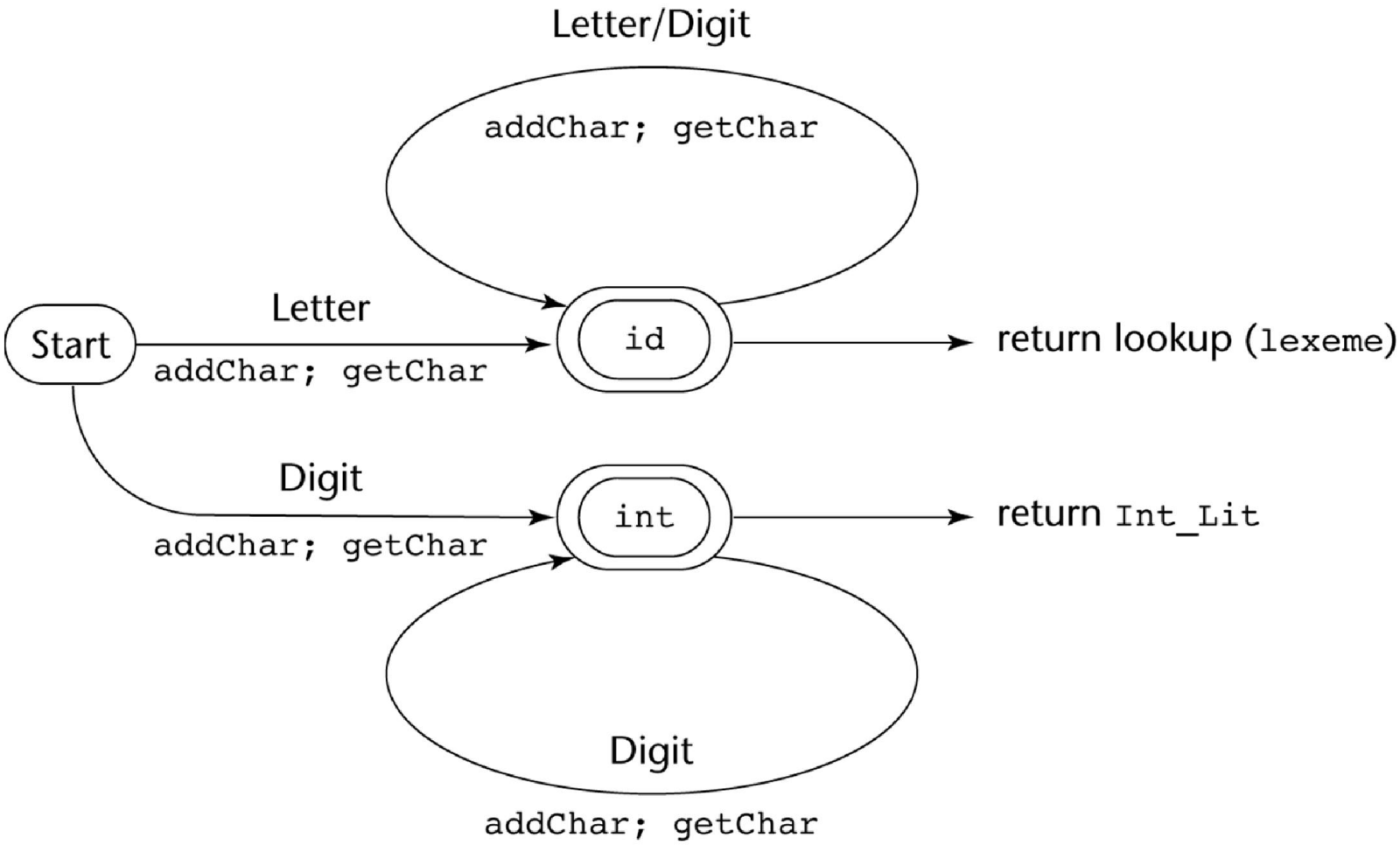
- Ayrılmış kelimeler (ör. if, while vs.) isimlerle birlikte tanınabilir. Sembol tablosunu önceden ayrılmış kelimelerle ilklemek şartı ile.

Sözcük Analizi ...

- Faydalı fonksiyonar:
  - **`getChar`** – bir sonraki karakteri girdiden alır, **`nextChar`** global değişkenine koyar, sınıfını belirler ve onu da **`charClass`** değişkenine koyar
  - **`addChar`** – **`nextChar`**'daki karakteri **`lexeme`** dizisinin sonuna ekler
  - Lookup() – sembol tablosuna bakarak **`lexeme'`**deki sözcüğün sembol tablosundaki adresini ve jetonunu verir (<sembolTablosuAdresi, jeton>). **`lexeme`**'deki sözcük sembol tablosunda yoksa, önce onu ID jetonu ile tabloya koyar.

- Insert(): argümanını sembol tablosuna yerleştirir, yerleştirilen yerin adresini verir

Durum geçiş diyagramı



Sözcük Analizi ...

```
/* Global değişkenler */

int charClass;
char lexeme [100];
char nextChar;
int lexLen;
#define LETTER = 0;
#define DIGIT = 1;
#define UNKNOWN = -1;
```

Sözcük Analizi ...

```
int lex() {
  lexLen = 0;
  static int first = 1;
/* ilk kez çağrılıyorsa birinci karakeri oku */
  if (first) {
    getChar();
    first = 0;
  }
  getNonBlank();
  switch (charClass) {

 /* Kullanıcı tanımlı isimleri ve ayrılmış sözcükleri tanı*/
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER || charClass == DIGIT){
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
          ...
```

Sözcük Analizi ...

```
...
/* Parse integer literals */
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return <insert(lexeme), INT_LIT>;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

Ayrıştırma (parsing) problemi

- Ayrıştırıcının amacı, ona verilen program için
  - Bütün sentaks hatalarını bulmak, her hata için açıklayıcı bilgi vermek ve hata olmamış gibe programın geri kalanını işlemeye devam edebilmek
  - Ayrıştırma ağacı, veya en azından ağaç oluşturulabilecek bilgiyi üretmek

Ayrıştırma (parsing) problemi ...

- İki türlü ayrıştırıcı var
  - *Yukarıdan aşağı* – ayrıştırma ağacı kökten başlayarak aşağıya doğru üretilir
    - Sırası soldan türeme ile ayni
    - Ayrıştırma ağacı önziyaret (preorder) sırası ile üretilir veya izlenir (üzerinden geçilir)
  - *Aşağıdan yukarı*– ayrıştırma ağacı yapraklardan köke doğru üretilir veya izlenir
    - Sırası "geriye doğru sağdan türeme" ile ayni
- Ayrıştırıcının faydalı olması için sadece bir tek jetonla ne yapacağına karar verbilmesi gerekir

Ayrıştırma (parsing) problemi ...

- Yukarıdan-Aşağı ayrıştırıcılar
  - xAα cümlemsisi için, ayrıştırıcı bir sonraki jetona bakarak A'nın hangi sağ tarafı ile değiştirilmesi gerektiğine karar verebilmesi gerekir
- En yaygın yukarıdan-aşağı ayrıştırıcılar:
  - Özyinelemeli iniş – gramerden elde edilen program (her nonterminal için bir fonksiyon)
  - LL ayrıştırıcılar – tablolu implementasyon

LL ayrıştırma algoritması

- Yığıt kullan
- Yıgıtın içinde önce başlangıç sembolü olsun
- Yığıtın en üst sembolü girdideki karakter ile ayni olan bir terminal ise, yığıttan at ve girdi işaretçisini ilerlet
- Yığıtın en üst sembolü bir nonterminal ise, girdideki karakteri ilk sembol olarak üretebilen bir sağ taraf ile degiştir
- Soldan türeme ile ilişkisi: tüketilmiş girdi + yığıt içeriği = türemedeki cümlemsi

LL Yukarıdan aşağı ayrıştırma örneği

- S -> T U
- T -> A B | e S
- U -> C D | f T
- A -> a
- B -> b
- C -> c
- D -> d
- Girdi: abcd

Ayrıştırma (parsing) problemi ...

- Ayrıştırmanın karmaşıklığı
  - *Herhangi* bir belirsiz olmayan gramer için çalışan ayrıştırıcılar verimsizdir (n büyüklüğünde girdi için $O(n^3)$)
  - Derleyiciler ancak belli gramerleri kullanabilen, ama O(n) zamanda çalışan ayrıştırıcılar kullanırlar.

Özyinelemeli-iniş ayrıştırıcıları

- Her nonterminal için o nonterminalin ürettiği stringleri ayrıştıran bir fonksiyon var.
- GBNF özyinelemeli-iniş ayrıştırıcıları için ideal, çünkü nonterminal sayısını azaltıyor

Özyinelemeli-iniş ayrıştırıcıları...

- Basit ifade grameri:

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

Özyinelemeli-iniş ayrıştırıcıları...

- `Lex` adında bir sözcük analizcisi varsayıyoruz. Bir sonraki jetonu `nextToken` un içine koyar.
- Kuralda tek sağ taraf varsa:
  - Sağ taraftaki her terminal için, bir sonraki jetona eşitse, devam. Aksi halde hata ver.
  - Sağ taraftaki her nonterminal için ayni isimdeki fonksiyonu çağır.

Özyinelemeli-iniş ayrıştırıcıları...

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

void expr() {


 term();
  ...
```

Özyinelemeli-iniş ayrıştırıcıları...

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

- Kural: Her fonksiyon bir sonraki jetonu `nextToken` içinde bırakır

Özyinelemeli-iniş ayrıştırıcıları...

- Bir nonterminalin birden çok sağ tarafı varsa, hangi sağ tarafın seçileceğini bir sonraki jeton belirler. Sağ taraflardan tam olarak bir tanesinin *ilk sembol* olarak bir sonraki jetonu üretebilmesi gerekir. Bu sembolü üretebiln sağ taraf, seçilen sağ taraftır.
- Hiçbir sağ taraf üretemiyorsa, veya birden çok sağ taraf üretebiliyorsa, bu bir hatadır.

Özyinelemeli-iniş ayrıştırıcıları...

```
/* <factor> -> id  |  (<expr>)  */

 void factor() {

   if (nextToken) == ID_CODE)

     lex();
```

Özyinelemeli-iniş ayrıştırıcıları...

```
 else if (nextToken == LEFT_PAREN_CODE) {
    lex();
    expr();
   if (nextToken == RIGHT_PAREN_CODE)
      lex();
    else
      error();
  }
  else error();
}
```

Özyinelemeli-iniş ayrıştırıcıları...

- The LL Gramer sınıfı
  - Sol özyineleme problemi
    - Gramerde direkt veya endirekt olarak özyineleme varsa, yukarıdan-aşağı bir ayrıştırıcı için kullanılamaz.
      - Sol özyinelemeyi kaldırabiliriz.

      Her A nonterminali için

      1. A-kurallarını $A \rightarrow A\alpha_1 \mid \ldots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$

      Şeklinde grupla, şöyle ki hiçbir β A ile başlamasın

      2. Orijinal A kurallarını aşağıdakilerle değiştir.

      $A \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_n A'$

      $A' \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_m A' \mid \epsilon$

Özyinelemeli-iniş ayrıştırıcıları...

- Gramerlerin yukarıdan-aşağı ayrıştırıcı için kullanımını engelleyen ikinci özellik, sağ tarafların "ayrık" olmamaları
  - Girdideki ilk jetona bakarak hangi sağ tarafı seçmemiz gerektiğini bulamayız
  - Tanım: İLK(α) = {a | α =>* aβ }

    (α =>* ε, ise ε İLK(α) içindedir)

Özyinelemeli-iniş ayrıştırıcıları...

- Karşılılı ayrık olma testi:
  - Her nonterminal A için, ikiden fazla sağ tarafı varsa, $A \rightarrow \alpha_i$ ve $A \rightarrow \alpha_j$ için aşağıdaki doğru olmalı:

$$\text{İLK}(\alpha_i) \cap \text{İLK}(\alpha_j) = \phi$$

- Örnekler:

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

$$D \rightarrow a \mid aB$$

Özyinelemeli-iniş ayrıştırıcıları...

- Sol çarpanlama (left factoring) bu problemi çözebilir.

$A \rightarrow a \mid a\ B$

yerine

$A \rightarrow a\ R$

$R \rightarrow \varepsilon \mid B$

Aşağıdan-yukarı ayrıştırma

- Sağdan türemiş cümlemsi α için, α nın hangi alt dizininin hangi kuralın sağ tarafı olduğunu bul. Bulunan sağ taraf sol tarafla değiştirildiğinde bir önceki sağdan türemiş cümlemsi elde edilmeli.
- En yaygın aşağıdan-yukarı algoritmaları LR ailesinden
- Yığıt implementasyonu: yığıt içeriği + tüketilmemiş girdi = sağdan türemiş cümlemsi (türemeyi sondan başa doğru okumak gerekiyor)

Aşağıdan-yukarı ayrıştırma örneği

- S -> T U
- T -> A B
- U -> C D
- A -> a
- B -> b
- C -> c
- D -> d
- Sağdan türeme: S => T U => T C D => T C d => T c d => A B c d => A b c d => a b c d

Aşağıdan-yukarı ayrıştırma …

- Tanım: β, ancak ve ancak aşağıdaki doğru ise $\alpha\beta w$ nın **tutacağıdır:**

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

Aşağıdan-yukarı ayrıştırma …

- İt-İndirge Algoritmaları
  - İndirgeme, yığıt üzerindeki tutacağı sol tarafı ile değiştirmedir.
  - İtme bir sonraki jetonu ayrıştırma yığıtı üzerine koymadır

Aşağıdan-yukarı ayrıştırma …

- LR ayrıştırıcıların avantajları:
  - Programlama dilleri için kullanılan hemen hemen tüm gramerler için kullanılabiliriler
  - Diğer aşağıdan-yukarı ayrıştırıcılardan daha kapsamlı bir gramer grubuyla çalışabilirler, ama onlar kadar verimli çalışırlar
  - Hataları mümkün olan en erken zamanda yakalarlar
  - LR sınıfındaki gramerler LL sınıfındakilerin üst kümesidir.

Aşağıdan-yukarı ayrıştırma …

- LR ayrıştırıcısının durumu LR konfigürasyonu ile gösterilir.

$(S_0X_1S_1X_2S_2…X_mS_m, a_ia_{i+1}…a_n\$)$

Yığıt içeriği: $S_0X_1S_1X_2S_2…X_mS_m$

$S_0 , S_1 , S_2… , S_m$: durumlar

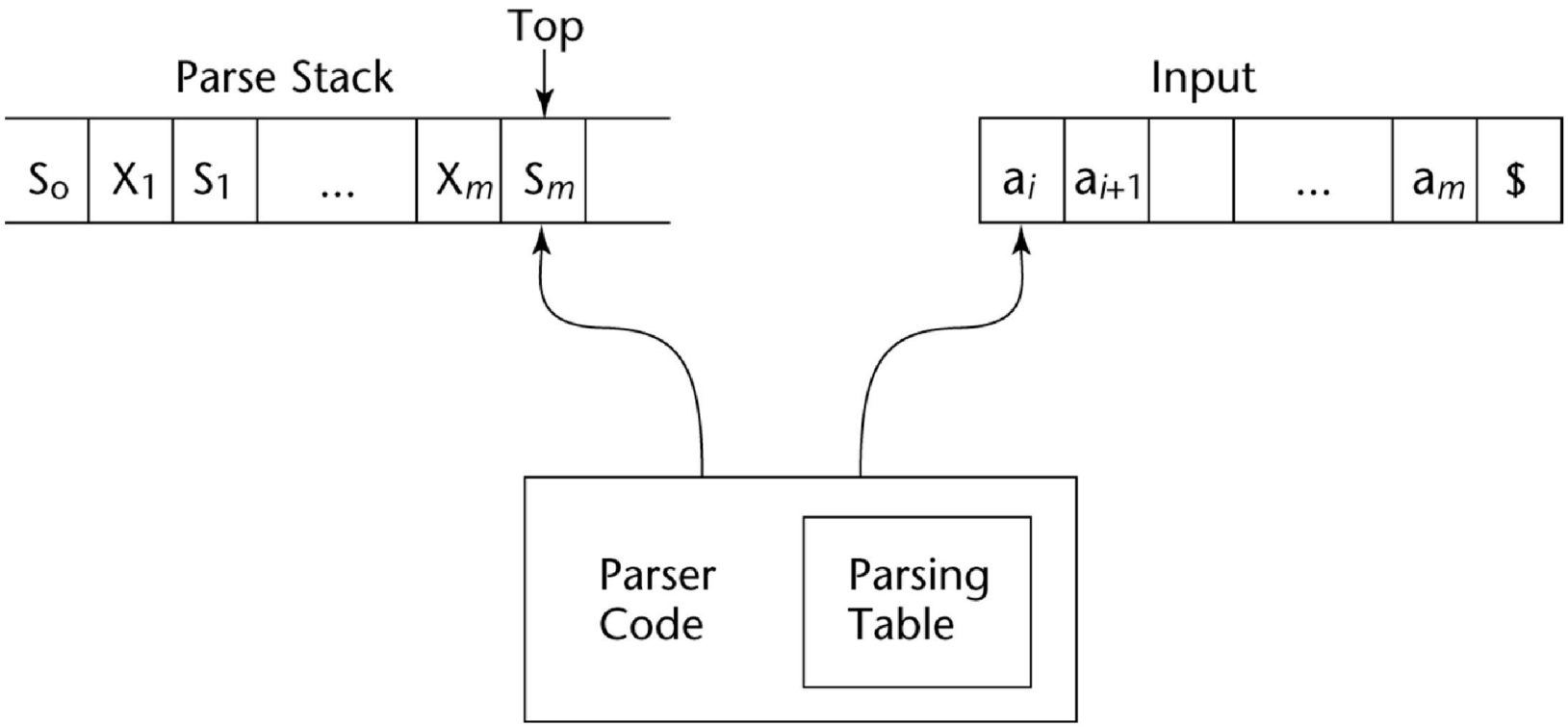
$X_1, X_2, …, X_m$: terminaller ve nonterminaller

Tüketilmemiş girdi: $a_ia_{i+1}…a_n$

Aşağıdan-yukarı ayrıştırma …

- LR ayrıştırıcılar tablo kullanarak çalışırlar. Tablonun iki kısmı var: ACTION ve GOTO
  - ACTION tablosu ayrıstırıcı durumu ve bir sonraki jetona göre ne yapılması gerektiğini söyler
    - Sıra: durum adı, Kolon: terminal
  - GOTO tablosu indirgeme (reduce) işleminden sonra hangi durumu yığıt üzerine koyacağımızı söyler
    - Sıra: durum adı, Kolon: nonterminal

Aşağıdan-yukarı ayrıştırma ...



Aşağıdan-yukarı ayrıştırma ...

- İlk konfigürasyon: $(S_0, a_1 \ldots a_n\$)$
- Ayrıştırıcı hareketleri:
  - ACTION$[S_m, a_i]$ = Shift(S) ise, bir sonraki konfigürasyon :

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

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

Aşağıdan-yukarı ayrıştırma …

- Ayrıştırıcı hareketleri …:
  - ACTION[S_m, a_i] = Accept ise (kabul), ayrıştırma bitmiştir, hata yok.
  - ACTION[S_m, a_i] = Error ise, ayrıştırıcı hata mesajı verir.

Aşağıdan-yukarı ayrıştırma örnegi

1. E -> E + T
2. E -> T
3. T -> T * F
4. T -> F
5. F -> (E)
6. F -> id

Ayrıştırılacak girdi: id + id * id

LR Ayrıştırma Tablosu

	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			

Ayrıştırıcı könfigürasyonları

Yığıt	Girdi	Hareket
0	id+id*id$	Shift 5
0id5	+id*id$	Reduce 6 (Goto [0,F])
0F3	+id*id$	Reduce 4 (Goto[0,T])
0T2	+id*id$	Reduce 2 (Goto[0,E])
0E1	+id*id$	Shift 6
0E1+6	id*id$	Shift 5
0E1+6id5	*id$	Reduce 6 (Goto[6,F])
0E1+6F3	*id$	Reduce 4 (Goto[6,T])
0E1+6T9	*id$	Shift 7
0E1+6T9*7	id$	Shift 5
0E1+6T9*7id5	$	Reduce 6 (Goto[7,F])
0E1+6T9*7F10	$	Reduce 3 (Goto[6,T])

Ayrıştırıcı könfigürasyonları ...

Yığıt	Girdi	Hareket
0E1+6T9	$	Reduce 1(Goto[0,E])
0E1	$	Accept

Aşağıdan-yukarı ayrıştırma …

- Belli bir gramer için ayrıştırıcı tabloalrı üreten araçlar vardır (ör: **`yacc, bison`**)

Özet

- Sentaks analizi dil implementasyonun sıradan bir işidir
- Sözcük analizcisi karakter gruplarını daha soyut nesnelere sınıflandıran bir desen uydurucudur (pattern matcher)
- Ayrıştırıcı, hataları bulur ve ayrıştırıcı ağacı üretir. İki tür ayrıştırıcı vardır:
  - Yukarıdan aşağı
    - Yığıt ile çalışan
    - Özyineleme-inen
  - Aşağıdn yukarı
    - İt-indirge ayrıştırıcıları