BMB214 Programlama Dilleri Prensipleri

Ders 4. Sözcüksel (Lexical) ve Sözdizimsel (Syntax) Analiz

Erdinç Uzun
Tekirdağ Namık Kemal Üniversitesi
Bilgisayar Mühendisliği Bölümü

Konular

- Sözcüksel Analiz
- Ayrıştırma (Parsing)
  - Özyineleme Kökenli Ayrıştırma
  - Aşağıdan Yukarıya Ayrıştırma

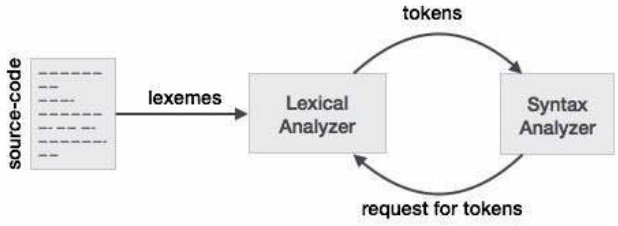
Giriş

- 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…

- 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…



- 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.

Giriş…

- 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…

- Sözcüksel (lexical) ve sözdizimi (syntax) analizini ayırmanın nedenleri:
  - Sadelik (Simplicity) – Sözcüksel analiz teknikleri sözdizimi analizinden daha az karmaşıktır, bu nedenle sözcük analiz süreci ayrı ise, ayrıştırıcı daha basit olabilir. Ayrıca sözcüksel analizin alt düzey ayrıntılarını sözdizim analizcisinden kaldırmak, sözdizimi analizcisini hem daha küçük hem de daha az karmaşık hale getirir
  - Verimlilik (Efficiency) –Sözcüksel analiz toplam derleme zamanının önemli bir kısmını gerektirdiğinden sözdizimi analizörünü optimize etmek verimli değildir. Ayırma, bu seçici optimizasyonu kolaylaştırır
  - Taşınabilirlik (Portability) – Sözcüksel analizci girdi program dosyalarını okur ve sıklıkla bu girişi tamponlamayı içerdiğinden, platforma bağımlıdır yani taşınamayabilir. Bununla birlikte, sözdizimi çözümleyicisi platformdan bağımsız olabilir. Herhangi bir yazılım sisteminin makineye bağımlı bölümlerini izole etmek her zaman iyidir.

Sözcüksel (Lexical) Analiz

- Sözcüksel analizci karakter katarları için bir desen eşleştiricidir
  - Regular Expression
- Başka bir değişle, verilen bir karakter kalıbıyla eşleşen belirli bir karakter dizesinin alt dizesini bulmaya çalışır
- Teknik olarak, sözcüksel analiz sözdizimi analizinin bir parçasıdır. Hatta ön ucu olarak değerlendirlebilir.
- Sözcüksel analizci, en düşük düzeyde program yapısında söz dizimi analizi yapar
- Birbirine ait olan kaynak programın alt dizelerini tanımlar – lexemes (sözcükler)
  - 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).

Sözcüksel (Lexical) Analiz…
Regular Expressions (Düzenli İfadeler)

- 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 = [+ | -]
- Örneğin bir string içinde sayıları bulan regex:
  - ([^\d]|^)\d{1,2},\d{3}([^\d]|$)

Sözcüksel (Lexical) Analiz…

- 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 (Lexical) Analiz…

- 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.

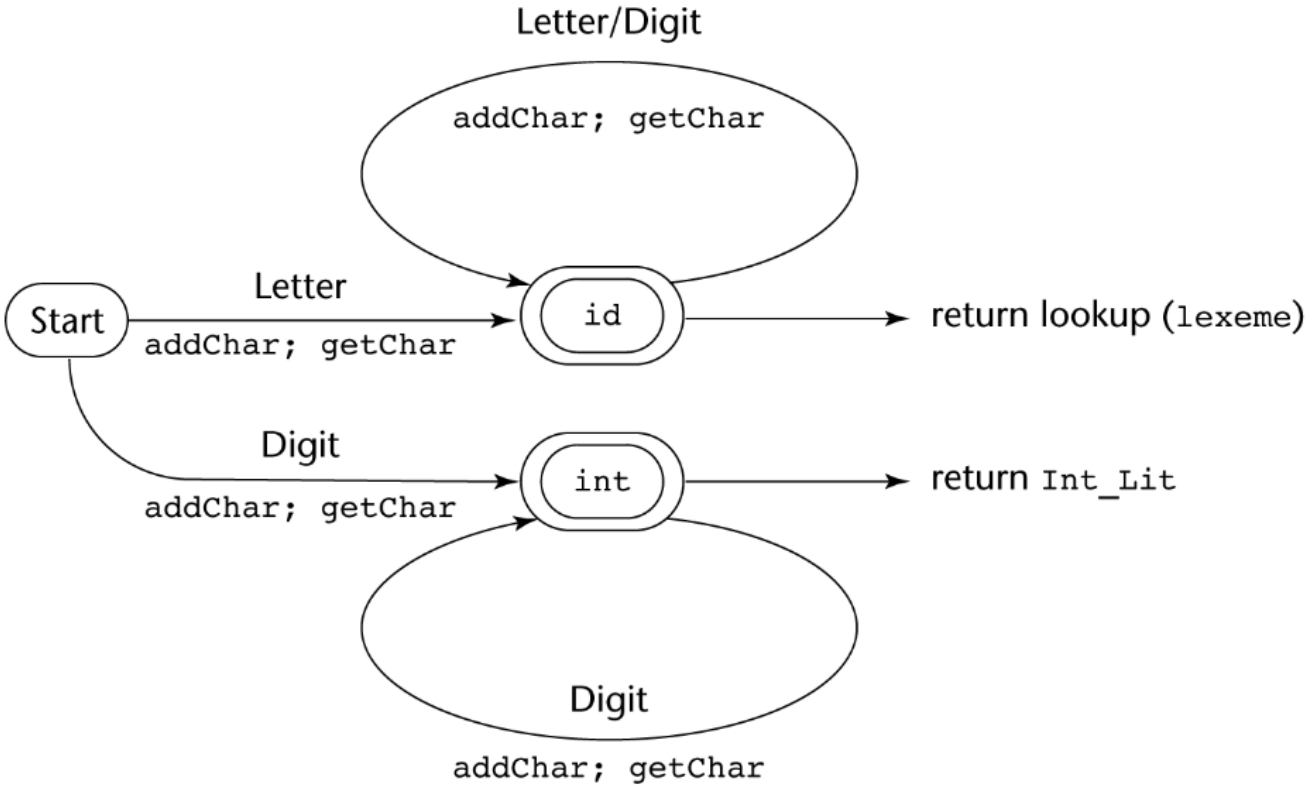
Durum diyagramı

- 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 Analiz Örneği

- Kullanışlı yardımcı altprogramlar(utility subprograms):
  - getChar–girdinin(input) sonraki karakterini alır, bunu nextChar içine koyar, sınıfını (class) belirler ve sınıfı(class)charClass içine koyar
  - addChar - nextChar'dan gelen karakteri lexemenin biriktirildiği yere koyar
  - arama(lookup)-lexemedeki katarın özgül sözcük(reservedword) olup olmadığını belirler(bir kod döndürür)

Durum Diyagramı



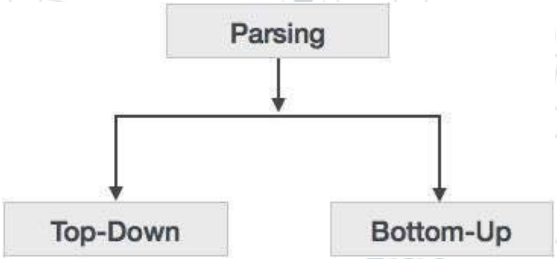
Durum Diyagramı Kaba Kodu

```
int lex() {
  switch (charClass) {
    case LETTER:
      addChar();
      getChar();
      while (charClass == LETTER ||
             charClass == DIGIT) {
        addChar();
        getChar();
      }
      return lookup(lexeme);
      break;
    case DIGIT:
      addChar();
      getChar();
      while (charClass == DIGIT) {
        addChar();
        getChar();
      }
      return INT_LIT;
      break;
  }  /* End of switch */
}  /* End of function lex */
```

Ayrıştırma (Parsing)

- Ayrıştırıcının (Parser) amaçları, bir girdi (input) programa verildiğinde:
  - Bütün sözdizim hatalarını (syntax errors) bulur; her birisi için, uygun bir tanılayıcı (diagnostic)mesaj üretir, ve hemen eski haline döndürür(recover)
  - Bir program için ayrıştırma ağacını (parse tree) veya en azından ayrıştırma ağacının izini (trace) üretir

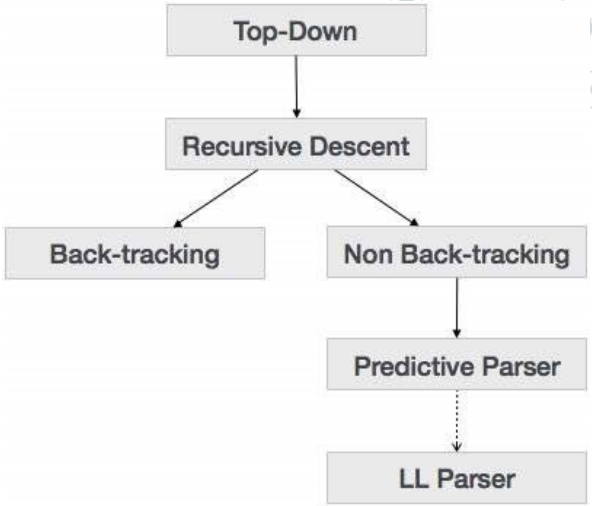
Ayrıştırıcıların (Parser) Kategorileri



- Ayrıştırıcıların iki kategorisi
  - Yukarıdan aşağı (Top down) - kökten başlayarak ayrıştırma ağacını üretir.
    - Sıra, en soldaki türetmedir (leftmost derivation)
    - Preorder bir şekilde ayrıştırma ağacını izler veya oluşturur
  - Aşağıdan yukarıya (Bottom up) - yapraklardan başlayarak ayrıştırma ağacını üretir.
    - Sıra, en sağdaki türetmenin (rightmost derivation) tersidir
- Ayrıştırıcılar, girdide (input) yalnızca bir token ileriye bakar

Top-down Parsers

- Yukarıdan aşağıya ayrıştırıcılar(Top-down parsers)
  - Bir xAα cümlesel formu verildiğinde, ayrıştırıcı (parser), sadece A'nın ürettiği ilk token'ı kullanarak, en sol türetmedeki (leftmost derivation)sonraki cümlesel formu (sentential form) elde etmek için doğru olan A-kuralını seçmelidir
  - Burada x:terminal sembol, A: nonterminal sembol, α: karma sembol
  - Başka bir deyişle, ayrıştırma ağacını üstten inşa eden ve girdi soldan sağa doğru okunan yukarıdan aşağıya bir ayrıştırma tekniğidir.
- En bilinen yukarıdan aşağıya ayrıştırıcı (top-down parser) algoritmaları:
  - Özyinemeli azalan(recursive-descent) - kodlanmış bir uygulama (Back-tracking)
  - Predictive Parser: LL ayrıştırıcılar(parser) - tabloya dayalı (table driven) uygulama (non back-tracking)



Bottom -up Parsers

- Bir, α sağ cümlesel formu (right sentential form) verildiğinde, a'nın sağ türetmede önceki cümlesel formu üretmesi için azaltılması gerekli olan, gramerde kuralın sağ tarafında (right-hand side) olan alt string'inin(substring) α ne olduğuna karar verir.
- En yaygın aşağıdan-yukarıya ayrıştırma (bottom-up parsing) algoritmaları LR ailesindedir

Ayrıştırma karmaşıklığı (Complexity of Parsers)

- Herhangi bir belirsiz - olmayan gramer (unambiguous grammar) için çalışan ayrıştırıcılar karmaşık ve verimsizdir: O(n3) [n girdi uzunluğu olmak üzere]
- Derleyiciler(compilers), sadece bütün belirsiz-olmayan gramerlerin (unambiguous grammars) bir altkümesi için çalışan ayrıştırıcıları kullanır, fakat bunu lineer sürede yapar: O(n) [n girdi uzunluğu olmak üzere]

Top-down Parsers
Özyineleme Kökenli Ayrıştırma (Recursive -Descent Parsing)

- Gramerdeki her bir non-terminal için o non-terminal tarafından üretilebilen cümleleri ayrıştıran bir alt program vardır
- EBNF özyineleme kökenli ayrıştırıcılara taban olması için ideal ölçüde uygundur; çünkü EBNF non-terminallerin sayısını minimize eder
- Basit bir expression için gramer:
  <expr> → <term> {(+ | -) <term>}
  <term> → <factor> {(* | /) <factor>}
  <factor> → id | int_constant | (<expr>)

Top-down Parsers
Özyineleme Kökenli Ayrıştırma…

- nextToken'ın içine bir sonraki token kodunu koyan lex isimli bir sözcüksel çözümleyicimiz olduğunu varsayalım
- Sadece bir sağ taraf (RHS) olduğu zamanki kodlama süreci:
  - Sağ taraftaki her bir terminal sembolü bir sonraki girdi token'ıyla karşılaştır; eğer eşleşirse devam et, eşleşmezse hata vardır
  - Sağ taraftaki her bir non-terminal sembol için onunla ilişkili olan ayrıştırma alt programını çağır

Top-down Parsers
Özyineleme Kökenli Ayrıştırma…

```
/* 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 program hataları yakalamaz
- Her ayrıştırma programı bir sonraki token'ınextToken'a bırakır

Top-down Parsers
Özyineleme Kökenli Ayrıştırma…

- 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

Top-down Parsers
Özyineleme Kökenli Ayrıştırma…

```
/* 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 */
```

Top-down Parsers
Özyineleme Kökenli Ayrıştırma…

```
/* 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 */
}
```

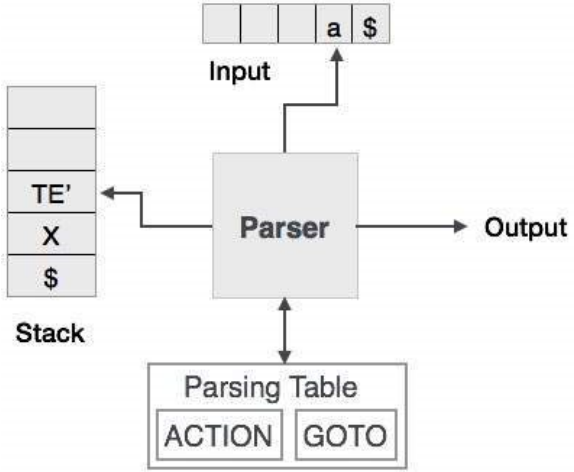
Top-down Parsers - Özyineleme Kökenli Ayrıştırma … Sözcüksel ve Sözdizimsel Analiz

(sum + 47) / total

```
Next token is: 25 Next lexeme is (
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 11 Next lexeme is sum
Enter <expr>
Enter <term>
Enter <factor>
Next token is: 21 Next lexeme is +
Exit <factor>
Exit <term>
Next token is: 10 Next lexeme is 47
Enter <term>
Enter <factor>
Next token is: 26 Next lexeme is )
Exit <factor>
Exit <term>
Exit <expr>
Next token is: 24 Next lexeme is /
Exit <factor>
Next token is: 11 Next lexeme is total
Enter <factor>
Next token is: -1 Next lexeme is EOF
Exit <factor>
Exit <term>
Exit <expr>
```

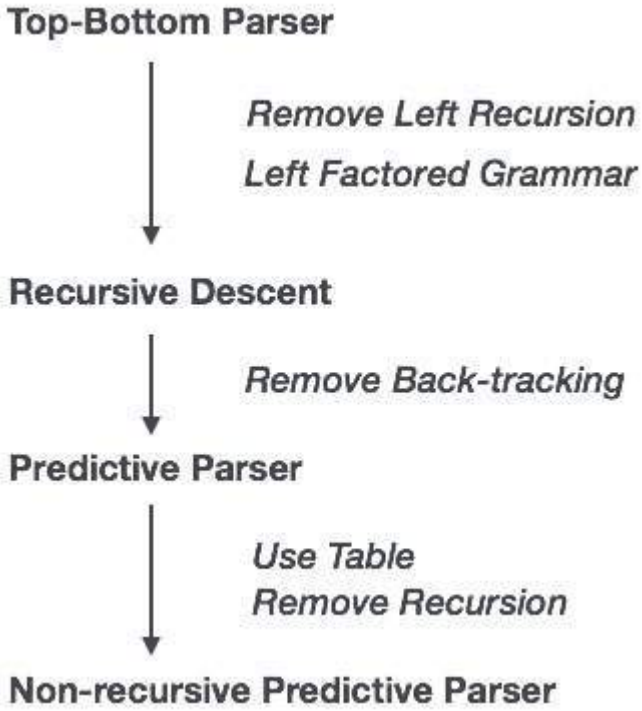
Top-down Parsers
Predictive Parser

- Predictive parser, girdi dizesini değiştirmek için hangi üretimin kullanılacağını tahmin etme yeteneğine sahip özyinelemeli bir iniş ayrıştırıcısıdır. Predictive parser, geri izleme (back-tracking) sorunu yaşamaz.
- Predictive parser, görevlerini yerine getirmek için bir sonraki giriş sembollerini gösteren ileriye dönük bir işaretçi (look-ahead pointer) kullanır. Ayrıştırıcıyı geri izlemesiz yapmak için, predictive parser dilbilgisine bazı kısıtlamalar getirir ve yalnızca LL (k) dilbilgisi olarak bilinen bir dilbilgisi sınıfını kabul eder.



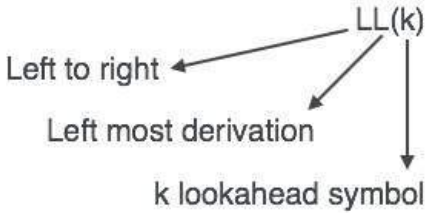
Top-down Parsers
Predictive Parser

- Predictive parser, girdiyi ayrıştırmak ve bir ayrıştırma ağacı oluşturmak için bir stack ve bir ayrıştırma tablosu (parsing table) kullanır. Hem stack hem de girdi, yığının boş olduğunu ve girdinin kullanıldığını belirtmek için bir bitiş sembolü $ içerir. Ayrıştırıcı, girdi ve stack elemanı kombinasyonu hakkında herhangi bir karar almak için ayrıştırma tablosuna başvurur.
- Özyineleme Kökenli Ayrıştırma, tek bir girdi örneği için seçilebilecek birden fazla üretime sahip olabilir, oysa predictive parser'da, her adımın seçilecek en fazla bir üretimi vardır. Girdi dizesiyle eşleşen üretimin olmadığı durumlar olabilir ve bu da ayrıştırma prosedürünün başarısız olmasına neden olur.



Top-down Parsers
LL Parser

- Bir LL Parser, LL gramer kabul eder. LL gramer, bağlamdan bağımsız dilbilgisinin bir alt kümesidir, ancak kolay uygulama sağlamak için basitleştirilmiş sürümü almak için bazı kısıtlamalar vardır. LL gramer'i, her iki algoritma, yani özyineleme kökenli (recursive descent) veya tablo tabanlı (table-driven) olarak uygulanabilir.
- LL ayrıştırıcısı LL (k) olarak belirtilir. LL (k) 'deki ilk L, girdiyi soldan sağa ayrıştırır, LL (k)' deki ikinci L, en soldaki türetmeyi (left-most derivation) temsil eder ve k'nin kendisi, önden bakma (look aheads) sayısını temsil eder. Genellikle k = 1 olduğundan LL (k) LL (1) olarak da yazılabilir.



Top-down Parsers
LL Parser

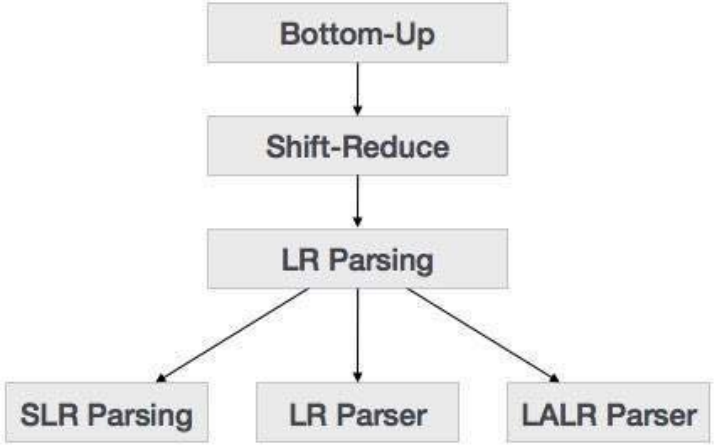
```
Input:
   string ω
   parsing table M for grammar G

Output:
   If ω is in L(G) then left-most derivation of ω,
   error otherwise.

Initial State : $S on stack (with S being start symbol)
   ω$ in the input buffer

SET ip to point the first symbol of ω$.

repeat
   let X be the top stack symbol and a the symbol pointed by ip.

   if X∈ Vt or $
      if X = a
         POP X and advance ip.
      else
         error()
      endif

   else /* X is non-terminal */
      if M[X,a] = X → Y1, Y2,... Yk
         POP X
         PUSH Yk, Yk-1,... Y1 /* Y1 on top */
         Output the production X → Y1, Y2,... Yk
      else
         error()
      endif
   endif
until X = $     /* empty stack */
```

Bir gramer G LL (1) 'dir, eğer A → α | β G'nin iki farklı ürünüdür:

- no-terminal için, hem α hem de β, a ile başlayan katarları (strings) türetir.
- en fazla α ve β'den biri boş katar türetebilir.
- β → t ise, α FOLLOW(A)'da bir terminal ile başlayan herhangi bir string türetmez.

Bottom -Up Parser (Aşağıdan Yukarıya Ayrıştırma)

- Bottom-Up Parser, bir ağacın yaprak düğümlerinden başlar ve kök düğüme ulaşana kadar yukarı yönde çalışır. Burada başlangıç sembolüne ulaşmak için bir cümleden başlayıp ardından üretim kurallarını tersine uyguluyoruz.
- Yandaki şekil farklı bottom-up parser yöntemleri gösterilmektir.



Bottom -Up Parser
Shift -Reduce Parser

- Shift-Reduce Parser, aşağıdan yukarıya ayrıştırma için kaydırma adımı ve azaltma adımı olarak bilinen iki benzersiz adım kullanır.
  - Shift step: Kaydırma adımı, giriş işaretçisinin kaydırılmış sembol olarak adlandırılan bir sonraki giriş sembolüne ilerlemesini ifade eder. Bu sembol stack'in üzerinde kaydırılır. Kaydırılan sembol, ayrıştırma ağacının tek bir düğümü olarak değerlendirilir.
  - Reduce step: Ayrıştırıcı, tam bir dilbilgisi kuralı (RHS) bulduğunda ve onu (LHS) ile değiştirdiğinde, bu, azaltma (reduce) adımı olarak bilinir. Bu, yığının tepesi bir handle içerdiğinde ortaya çıkar. Azaltmak için, handle'dan çıkan ve onu LHS terminal olmayan sembolle değiştiren yığın üzerinde bir POP işlevi gerçekleştirilir.

Bottom -Up Parsers
LR Parser

- LR parser yinelemeli olmayan, shift-reduce, aşağıdan yukarıya ayrıştırıcıdır. Geniş bir bağlamdan bağımsız gramer sınıfını kullanır ve bu da onu en verimli sözdizimi analizi tekniği yapar. LR parser, LR(k) ayrıştırıcıları olarak da bilinir; burada L, giriş akışının soldan sağa taramasını ifade eder; R, en sağdaki türetmenin tersine inşasını temsil eder ve k, kararlar almak için ileri dönük sembollerin sayısını gösterir.

Bottom -Up Parsers
LR Parser …

Bir LR parser oluşturmak için yaygın olarak kullanılan üç algoritma vardır:

- SLR(1) - Simple LR Parser:
  - En küçük gramer sınıfında çalışır
  - Az sayıda durum, dolayısıyla çok küçük tablo
  - Basit ve hızlı oluşur
- LR(1) - LR Ayrıştırıcı:
  - Tam LR (1) gramer seti üzerinde çalışır
  - Büyük tablo ve çok sayıda durum oluşturur
  - Yavaş oluşur
- LALR(1) - İleri Bakış (Look-Ahead) LR Ayrıştırıcı:
  - Orta düzey gramer üzerinde çalışır
  - Durum sayısı SLR(1)'deki ile aynıdır

Bottom -Up Parsers
LR Parser Kaba Kodu

```
token = next_token()

repeat forever
   s = top of stack

   if action[s, token] = “shift si” then
      PUSH token
      PUSH si
      token = next_token()

   else if action[s, token] = “reduce A::= β“ then
      POP 2 * |β| symbols
      s = top of stack
      PUSH A
      PUSH goto[s,A]

   else if action[s, token] = “accept” then
      return

   else
      error()
```

LL Parser vs. LR Parser

LL	LR
Leftmost derivation yapar.	Tersine rightmost derivation yapar.
Stack üzerindeki nonterminal kök ile başlar.	Stack üzerinde nonterminal kök ile biter.
Stack boş kaldığında biter	Bir boş stack ile başlar
Stack'i, beklenen şeyi belirtmek için kullanır.	Stack'i, zaten görüleni belirtmek için kullanır.
Ayrıştırma ağacını yukarıdan aşağıya oluşturur.	Builds the parse tree bottom-up.
Sürekli olarak stack bir nonterminal çıkarır (POP) ve karşılık gelen sağ tarafı iter (PUSH).	Stack'te sağ tarafını tanımaya çalışır, onu çıkarır(POP) ve karşılık gelen nonterminal'i iter (PUSH).
Non-terminal'leri genişletir.	Non-terminal'leri azaltır.
Stack'ten bir tane çıkardığında (POP) terminalleri okur.	Stack üzerine iterken (PUSH) terminalleri okur.
Ayrıştırma ağacını pre-order şekilde dolaşır.	Ayrıştırma ağacını post-order şekilde dolaşır.

https://www.tutorialspoint.com/compiler_design/compiler_design_bottom_up_parser.htm

LL Parser vs. LR Parser

LL	LR
Uygulaması kolay	Uygulaması zor
Önceki geliştirilen algoritmalardan az güçlü	Önceki geliştirilen algoritmalara göre güçlü
Tarihsel olarak çok tercih edilen çözüm	Yüksek performans (Linear, Bazı durumlarda $O(N^3)$ kötü performans durumları var.)
Geliştirmesi kolay	En iyi performansa sahip parser

Özyineleme Kökenli Ayrıştırma Örnekleri

- http://athena.ecs.csus.edu/~purohitj/parser.html
- http://ericbn.github.io/js-abstract-descent-parser/

- Javascript tabanlı örnekler, gramerlere dikkat!!!

Popüler Hazır Parser Üreteçleri (Parser Generators)

- https://en.wikipedia.org/wiki/Comparison_of_parser_generators
  - Kullandıkları algoritma/lar
  - Programlama Dili Çıktıları
  - Gramer notasyonu
- https://tomassetti.me/guide-parsing-algorithms-terminology/
- https://tomassetti.me/parsing-in-javascript/
  - https://chevrotain.io/docs/