

Course: Compiler Construction

Week 6: Top-Down Parsing

Lecturer: Martha Gichuki

Learning outcomes Week 6: Top-Down Parsing

At the end of the lecture, you will be able to:

- (i) Define Recursive Descent Parsing and Predictive Parsing
- (ii) Describe backtracking technique
- (iii) Describe lookahead set of terminals

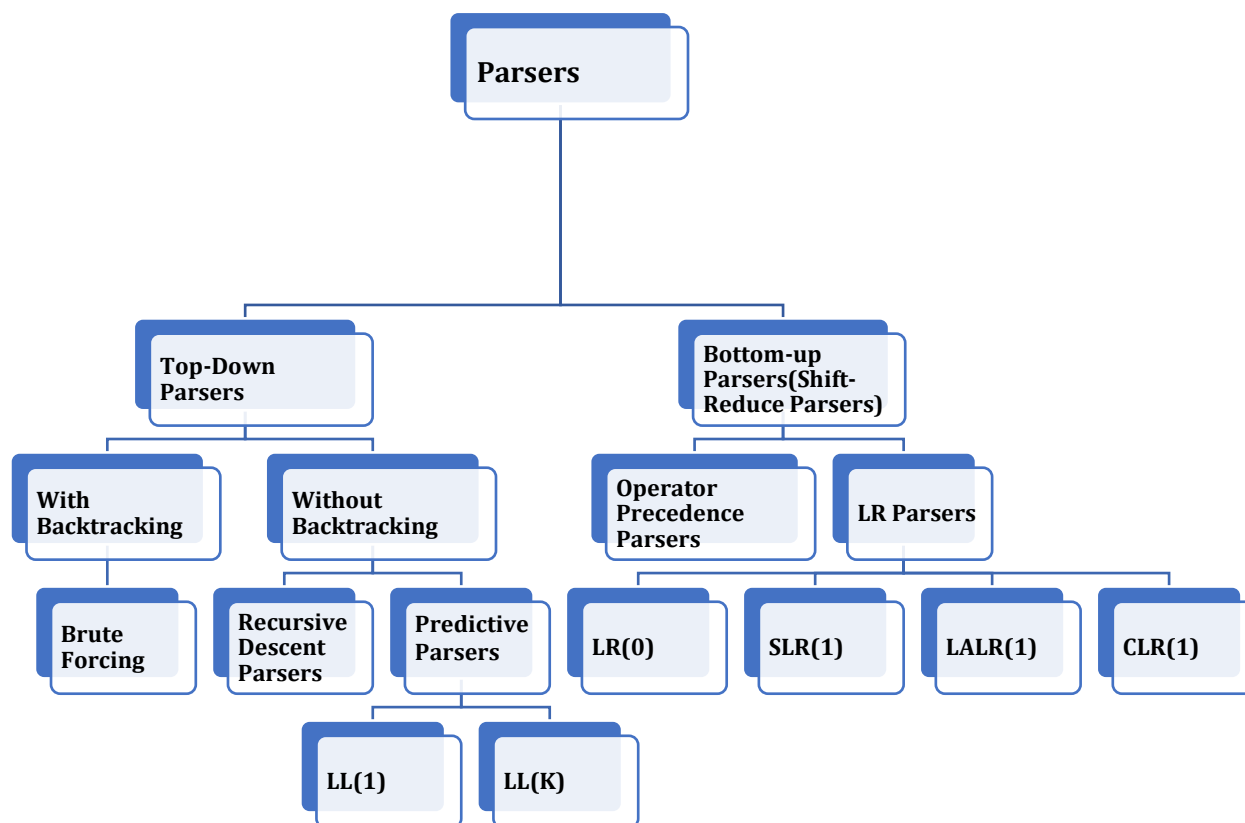
1. Introduction

1.1. Definition: Parsing

- ✓ Parsing is a convenient way to see how strings are derived from the start symbol.
- ✓ It is a way of analyzing a string or text into logical syntactic components
- ✓ Tokens are divided into parts and the relations between the parts can then be described¹

¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 86

1.2. Classification of Parsers²



2. Top-Down Parsing

This parsing strategy finds the derivation of the input string **from the start symbol** of the grammar³. There are two main approaches for Top-Down parsing:

- A. Recursive Descent Parsing
- B. Predictive Parsing

A. Recursive Descent Parsing⁴

- ✓ This is a common form of top-down parsing that uses a parsing technique, which recursively parses the input to make a parse tree from the top and the input is read from left to right.
- ✓ It is called recursive, as it uses recursive procedures to process the input.

² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 324

³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 86

⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 242

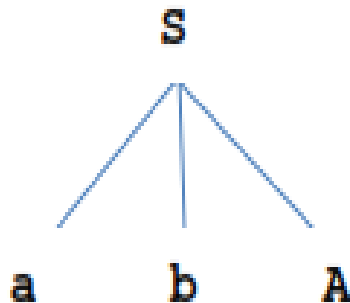
- ✓ The main idea is that **each non-terminal** in the grammar is **implemented by a procedure** in the program.
- ✓ Generally, these parsers consist of a set of mutually recursive routines that may require backtracking to create the parse tree.
- ✓ **Definition - Backtracking:** If one derivation of a production fails, the syntax analyzer restarts the process using **different rules** of the same production⁵.
- ✓ Backtracking technique may process the input string more than once to determine the right production.

Example: Consider the grammar:

$$S \rightarrow abA$$

$$A \rightarrow cd|c|e$$

- ✓ For the input stream **ab**, the recursive descent parser starts by constructing a parse tree representing $S \rightarrow abA$ as shown below⁶:

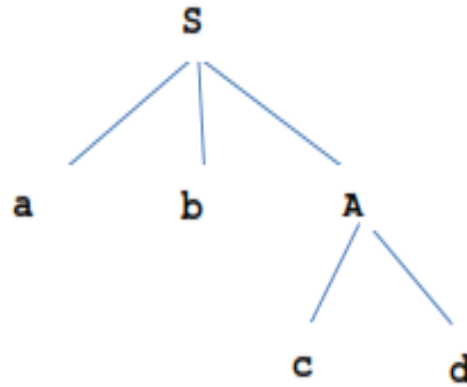


- ✓ The stream is then expanded with the production $A \rightarrow cd$ as shown below⁷

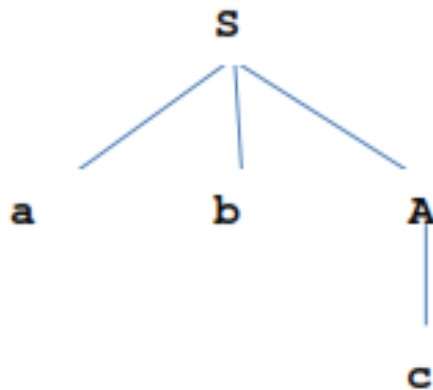
⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 244

⁶ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 244

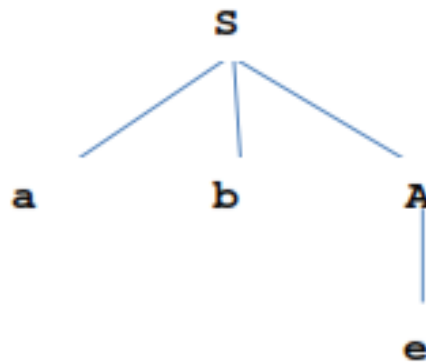
⁷ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 246



✓ Since it does not match ab , it backtracks and tries the alternative $A \rightarrow c$ as shown below⁸:



✓ However, the parse tree does not match ab ; the parser backtracks and tries the alternative $A \rightarrow e$ as shown below:-



✓ With this, it finds a match and thus parsing is completely successful⁹.

⁸ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 246

⁹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 247

- ✓ Generally, using recursive descent parsing, one can write a procedure for each non-terminal in the language definition:
 - (i) Where the **right-hand side** contains alternatives, the **next symbol to input** provides the **selector for a switch statement**, each **branch** of which represents **one alternative**¹⁰.
 - (ii) If **e** is an alternative, it becomes the **default** of the switch.
 - (iii) Where a **repetition occurs**, it can be implemented **iteratively**.
 - (iv) The sequence of actions within each branch, possibly one, consists of an inspection of the lexical token for each terminal, and a **procedure** will be **assigned to a local or global variable** as required by semantics of the language¹¹.

Example: The following is a recursive descent parser for the grammar shown below¹²:

- ✓ $T' \rightarrow T\$$
- ✓ $T \rightarrow R$
- ✓ $T \rightarrow aTc$
- ✓ $R \rightarrow \epsilon$
- ✓ $R \rightarrow bR$

parseT'() =

if next = 'a' or next = 'b' or next = '\$' then

parseT() ; match('\$')

else reportError ()

parseT() =

if next = 'b' or next = 'c' or next = '\$' then

parseR()

else if next = 'a' then

match('a') ; **parseT()** ; match('c')

else reportError()

parseR() =

if next = 'c' or next = '\$' then

(* do nothing *)

¹⁰ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 89

¹¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 242

¹² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 244

else if next = 'b' then

 match('b') ; parseR()

else reportError()¹³

B. Predictive Parsing

- ✓ Predictive parsing is a special form of recursive descent parsing in which the **current input token unambiguously determines the production to be applied at each step** i.e. they are *backtrack free*¹⁴.

Non-recursive Predictive – LL(k) Parsing

- ✓ *LL* Stands for: **Left to right scan; Left most derivation.**
- ✓ This is a parsing strategy which involves **building tables instead of writing code.**
- ✓ It is also possible to **automatically** create tables.
- ✓ These parsers scan over the input stream using a prefix of tokens so as to identify production to be applied, where ***k* is the length of the prefix**¹⁵.
- ✓ The language accepted by these parsers is called ***LL(k)***, where ***k* is the length of the prefix**. The length of the prefix to be considered will be **$k = 1$** ¹⁶.
- ✓ The construction of an ***LL(1)*** parser for the grammar $\langle V_N, V_T, P, S \rangle$ requires computation of the following properties:-
 - I. **FIRST(A) = {u | u ∈ V_T and A can derive a string starting with u}**¹⁷
 - II. **FOLLOW(A) = {u | u ∈ V_T and S can derive a string like Auβ}**
 - III. **FIRST(a) = {u | u ∈ V_T and a can derive a string like uβ}**

Recall that:-

- ✓ **FIRST (A)** is the set of terminals that **can start** a string derivable from A;
- ✓ **FOLLOW (A)** is the set of terminals that **can follow** an occurrence of A,
- ✓ **FIRST (a)** is the set of terminals that **can start** a string derivable from **a**¹⁸.

¹³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 89

¹⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 89

¹⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 132

¹⁶ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 226

¹⁷ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 227

¹⁸ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 227

3. LOOKAHEAD

- ✓ For a production rule, $A \rightarrow \alpha$, $LOOKAHEAD(A \rightarrow \alpha)$ is defined as the set of terminals which can **appear next in the input** when recognizing a production rule $A \rightarrow \alpha$.
- ✓ Thus, a production rule's LOOKAHEAD set **specifies the tokens which should appear next in the input before a production is applied**¹⁹.

3.1. To build $LOOKAHEAD(A \rightarrow \alpha)$:

- i). Put $FIRST(\alpha) - \{e\}$ in $LOOKAHEAD(A \rightarrow \alpha)$
- ii). If $e \in FIRST(\alpha)$

then put $FOLLOW(A)$ in $LOOKAHEAD(A \rightarrow \alpha)$

- ✓ A grammar G is $LL(1)$ **iff**, for each set of production, $A \rightarrow \alpha_1 | \alpha_2 | \dots | \alpha_n$

$LOOKAHEAD(A \rightarrow \alpha_1) LOOKAHEAD(A \rightarrow \alpha_2) \dots$

$LOOKAHEAD(A \rightarrow \alpha_n)$ are **all pairwise disjoint**²⁰

3.2. Facts about $LL(1)$ grammar:

- i). No left recursive grammar is $LL(1)$.
- ii). No ambiguous grammar is $LL(1)$.
- iii). Some languages have no $LL(1)$ grammar.
- iv). An **e -free grammar** where each alternative expansion for A begins with a distinct terminal is a **simple $LL(1)$ grammar**²¹.

Example 1: Given the grammar $S \rightarrow aS|a$, find whether it is $LL(1)$ grammar.

Solution:

- ✓ $FIRST(S) = \{a\}$

- ✓ Therefore, $LOOKAHEAD(S \rightarrow aS) = LOOKAHEAD(S \rightarrow a) = \{a\}$

Hence it is **not** $LL(1)$ grammar. **Note:** FOLLOW sets are not required since empty sets are not required in the grammar²².

¹⁹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

²⁰ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

²¹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87, 88

²² Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 87

Example 2: Consider the following grammar:

$$S \rightarrow aAa$$

$$S \rightarrow b$$

$$A \rightarrow ab$$

$$A \rightarrow e$$

Verify whether the grammar is $LL(1)$ or not.

Solution:

$$LOOKAHEAD(S \rightarrow aAa) = FIRST(a) = \{a\}$$

$$LOOKAHEAD(S \rightarrow b) = FIRST(b) = \{b\}$$

$$LOOKAHEAD(A \rightarrow ab) = FIRST(a) = \{a\}$$

$$LOOKAHEAD(A \rightarrow e) = FOLLOW(A) = \{a\}$$

Therefore, the grammar is **not $LL(1)$** since a is in **both** $A \rightarrow ab$ and $A \rightarrow e$ ²³.

3.3. $LL(1)$ Parse Table Construction Method 1 :

✓ Once the **first and follow states have been computed**, it is possible to construct $LL(1)$ parse table M that **maps pairs of non-terminals and terminals to production** using the following method:

Input: Grammar G

Output: Parsing table M ²⁴

Method:

1. \forall production $A \rightarrow \alpha$

$\forall a \in LOOKAHEAD A \rightarrow \alpha$

add $A \rightarrow \alpha$ to $M[A, a]$

2. Set each unidentified entry of M to error

3. If $\exists M[A, a]$ with **multiple entries** then the grammar is **not $LL(1)$** ²⁵.

²³ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 233

²⁴ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 225

²⁵ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 297

Example: Consider the following grammar²⁶:

$$S \rightarrow E$$

$$E \rightarrow TE'$$

$$E' \rightarrow +E|-E|e$$

$$T \rightarrow FT'$$

$$T' \rightarrow *T|/T|e$$

$$F \rightarrow id|num$$

The following is the **table of FIRST and FOLLOW sets²⁷**:

Symbol	FIRST	FOLLOW
S	{num, id}	{\$}
E	{num, id}	{\$}
E'	{+, -, e}	{\$}
T	{num, id}	{+,-,\$}
T'	{*, /, e}	{+,-,\$}
F	{num, id}	{+,-,*,/, \$}
Id	{id}	
num	{num}	
*	{*}	
/	{/}	
+	{+}	
-	{-}	

²⁶ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 250

²⁷ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 251

The following is the table of **LOOKAHEAD sets**: NB: The **\$ sign** is a special symbol for ‘*end of input*’²⁸.

Production	LOOKAHEAD
$S \rightarrow E$	{num, id}
$E \rightarrow TE'$	{num, id}
$E' \rightarrow +E$	{+}
$E' \rightarrow -E$	{-}
$E' \rightarrow e$	{ $\$$ }
$T \rightarrow FT'$	{num, id}
$T' \rightarrow *T'$	{*}
$T' \rightarrow /T$	{/}
$T' \rightarrow e$	{+,_, $\$$ }
$F \rightarrow id$	{id}
$F \rightarrow num$	{num}

²⁸ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 253

The following is the **parsing table**²⁹:

Symbol	Id	Num	+	-	*	/	\$
S	$S \rightarrow E$	$S \rightarrow E$					
E	$E \rightarrow TE'$	$E \rightarrow TE'$					
E'			$E' \rightarrow +E$	$E' \rightarrow -E$			$E' \rightarrow e$
T	$T \rightarrow FT'$	$T \rightarrow FT'$					
T'			$T' \rightarrow e$	$T' \rightarrow e$	$T' \rightarrow *T$	$T' \rightarrow /T$	$T' \rightarrow e$
F	$F \rightarrow id$	$F \rightarrow num$					

²⁹ Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007) page 250

Content Covered in Week 6: Top-Down parsing

At the end of the lecture, we were able to:

- (i) Define Recursive Descent Parsing and Predictive Parsing
- (ii) Describe backtracking technique
- (iii) Describe lookahead set of terminals

Course Text Books

1. Compilers: Principles, Techniques and Tools, Aho, Lam, Sethi, and Ullman; Addison- Wesley Pub Co, ISBN: 0201100886 (2007))
2. Compiler Construction: Principles and practices; Kenneth C. Loudon; Cengage Learning; 1st edition, ISBN-10: 0534939724 (1997)
3. Basics of Compiler Design: Torben Mogensen; DIKU University of Copenhagen Universitetsparken 1 DK-2100 Copenhagen DENMARK (2007)
4. Engineering a Compiler: 2nd edition; Keith D. Cooper and Linda Torczon; Morgan Kaufmann Publishers ISBN: 978-0-12-088478-0 (2003)
5. Compiler Design: Santanu Chattopadhyay; PHI Learning Publishers, ISBN 812032725X, 9788120327252 (2005)