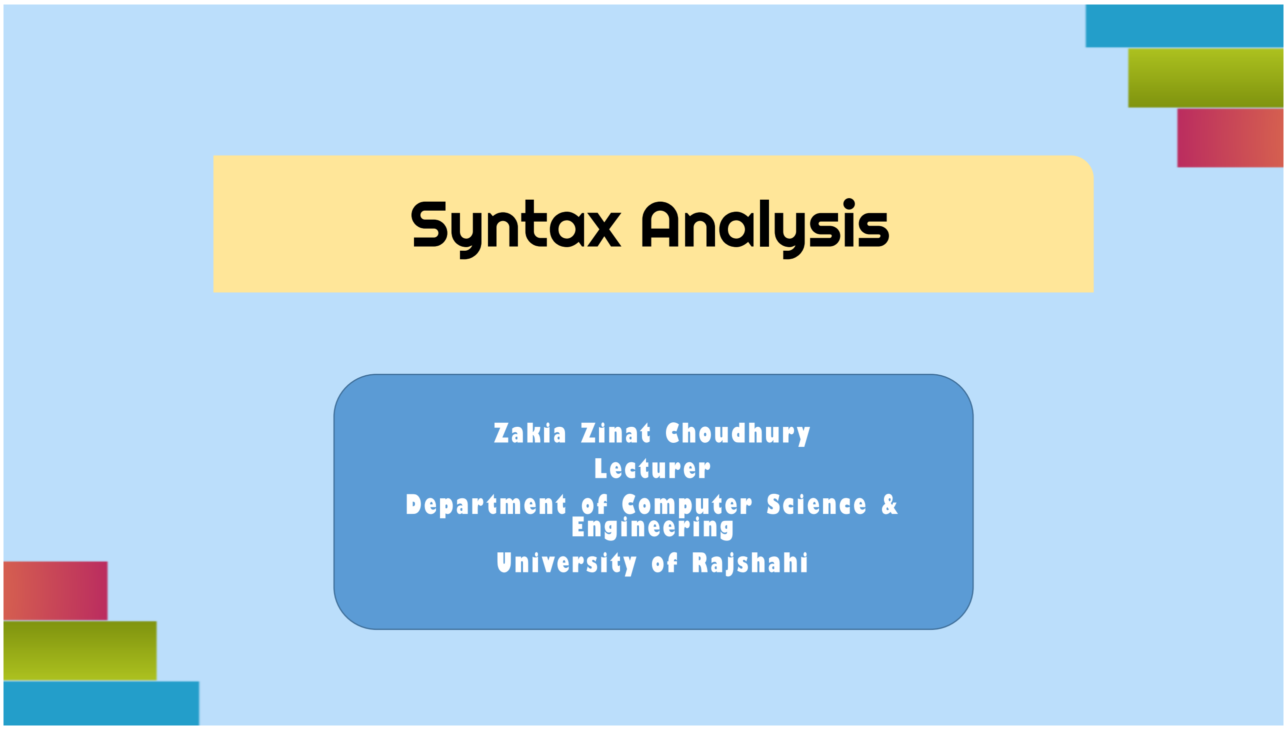
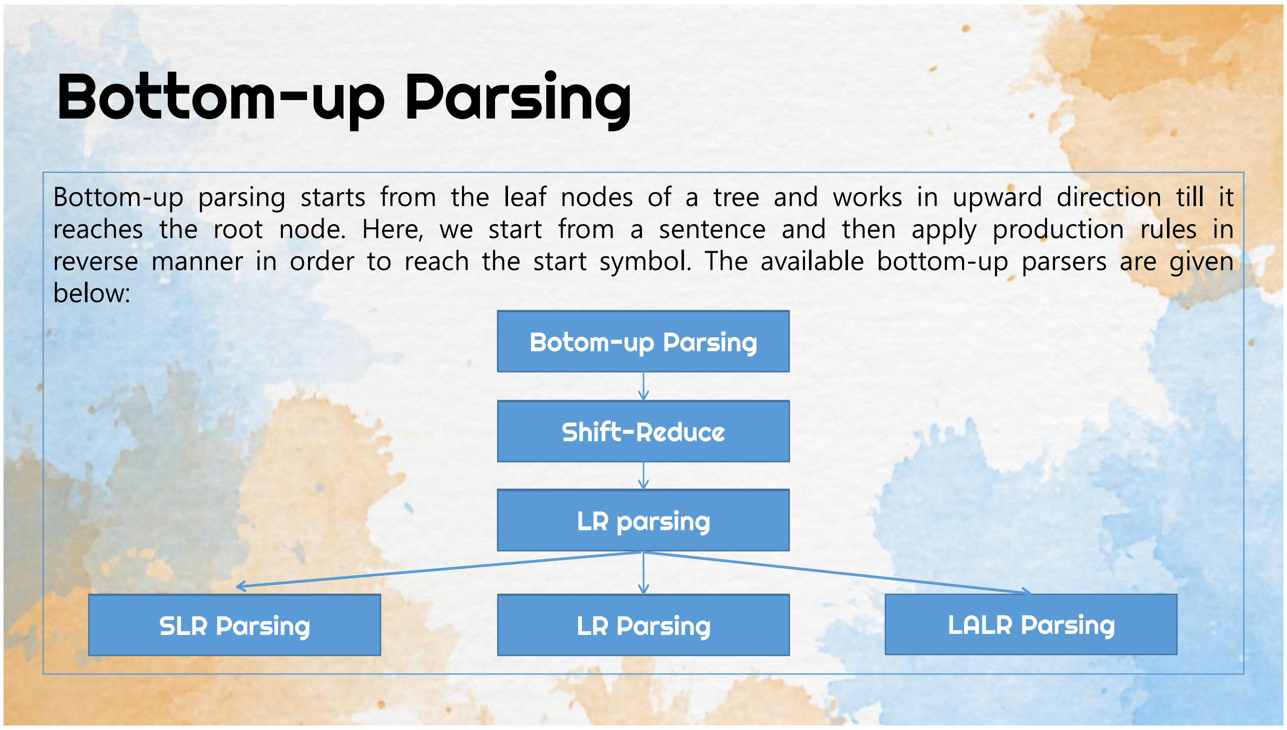


Syntax Analysis

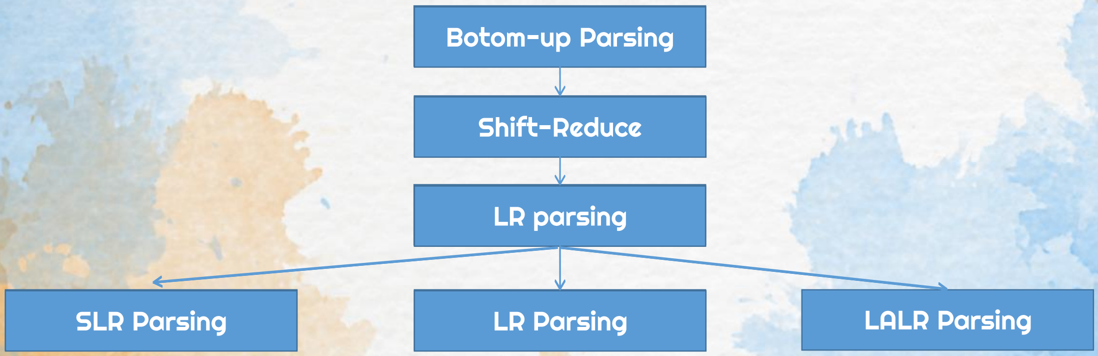
Zakia Zinat Choudhury
Lecturer
Department of Computer Science & Engineering
University of Rajshahi

1



Bottom-up Parsing

Bottom-up parsing starts from the leaf nodes of a tree and works in upward direction till it reaches the root node. Here, we start from a sentence and then apply production rules in reverse manner in order to reach the start symbol. The available bottom-up parsers are given below:



2

Reductions

- At each reduction step, a specific substring matching the body of a production is replaced by the nonterminal at the head of that production.

- The key decisions during bottom-up parsing are about when to reduce and about what production to apply, as the parse proceeds.

3

Handle Pruning

- Bottom-up parsing during a **left-to-right** scan of the input constructs a rightmost derivation in reverse.

- Informally, a **"handle"** is a substring that matches the body of a production, and whose reduction represents one step along the **reverse of a rightmost derivation**.

4

Shift-Reduce Parsing

- Shift-reduce parsing is a form of bottom-up parsing in which a stack holds grammar symbols, and an input buffer holds the rest of the string to be parsed.
- We use $ to mark the bottom of the stack and the right end of the input.

5

Shift-Reduce Parsing

- Initially, the stack is empty, and the string w is on the input, as follows:

Stack	Input
$	w$

- During a left-to-right scan of the input string, the parser shifts zero or more input symbols onto the stack, until it is ready to reduce a string β of grammar symbols on top of the stack. It then reduces β to the head of the appropriate production. The parser repeats this cycle until it has detected an error or until the stack contains the start symbol and the input is empty:

Stack	Input
$S	$

- Upon entering this configuration, the parser halts and announces successful completion of parsing.

6

Shift-Reduce Parsing

- While the primary operations are shift and reduce, there are actually four possible actions a shift-reduce parser can make: (1) shift, (2) reduce, (3) accept, and (4) error.
  - **Shift:** Shift the next input symbol onto the top of the stack.
  - **Reduce:** The right end of the string to be reduced must be at the top of the stack. Locate the left end of the string within the stack and decide with what nonterminal to replace the string.
  - **Accept:** Announce successful completion of parsing.
  - **Error:** Discover a syntax error and call an error recovery routine.

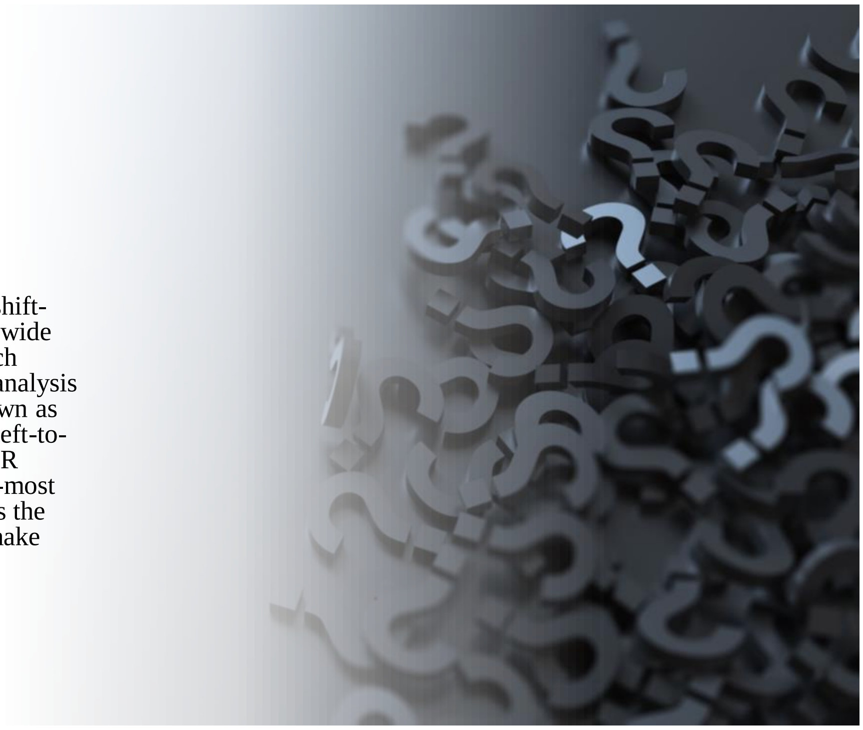
7

Conflicts During Shift-Reduce Parsing

- There are context-free grammars for which shift-reduce parsing cannot be used. Every shift-reduce parser for such a grammar can reach a configuration in which the parser, knowing the entire stack contents and the next input symbol, cannot decide whether to shift or to reduce (a shift/reduce conflict),or cannot decide which of several reductions to make (a reduce/reduce conflict).

- Another common setting for conflicts occurs when a handle is used, but the stack contents and the next input symbol are insufficient to determine which production should be used in a reduction.

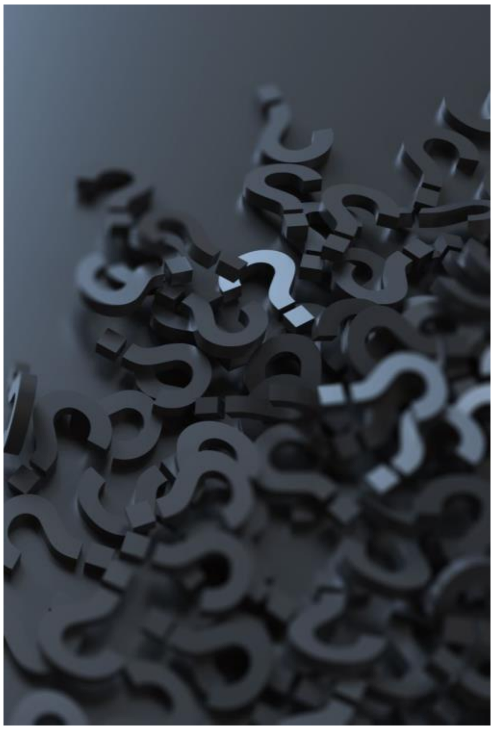
8



LR Parser

The LR parser is a non-recursive, shift-reduce, bottom-up parser. It uses a wide class of context-free grammar which makes it the most efficient syntax analysis technique. LR parsers are also known as LR(k) parsers, where L stands for left-to-right scanning of the input stream; R stands for the construction of right-most derivation in reverse, and k denotes the number of lookahead symbols to make decisions.

9



Why LR Parsers?

LR parsing is attractive for a variety of reasons:

- LR parsers can be constructed to recognize virtually all programming language constructs for which context-free grammars can be written. Non- LR context-free grammars exist, but these can generally be avoided for typical programming-language constructs.
- The LR-parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other, more primitive shift-reduce methods.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

10

LL vs LR

LL	LR
Does a leftmost derivation.	Does a rightmost derivation in reverse.
Starts with the root nonterminal on the stack.	Ends with the root nonterminal on the stack.
Ends when the stack is empty.	Starts with an empty.
Uses the stack for designating what is still to be expected.	Uses the stack for designating what is already seen.
Builds the parse tree top-down.	Builds the parse tree bottom-up.
Continuously pops a nonterminal off the stack and pushes the corresponding right hand side.	Tries to recognize a right-hand side on the stack, pops it, and pushes the corresponding nonterminal.
Expands the non-terminals.	Reduces the non-terminals.
Reads the terminals when it pops one off the stack.	Reads the terminals while it pushes them on the stack.

11

Error Recovery

A program may have the following kinds of errors at various stages:

- Lexical : name of some identifier typed incorrectly
- Syntactical : missing semicolon or unbalanced parenthesis
- Semantical : incompatible value assignment
- Logical : code not reachable, infinite loop

12

Error Recovery

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

Panic mode

When a parser encounters an error anywhere in the statement, it ignores the rest of the statement by not processing input from erroneous input to delimiter, such as semi-colon. This is the easiest way of error-recovery and also, it prevents the parser from developing infinite loops.

Statement mode

When a parser encounters an error, it tries to take corrective measures so that the rest of inputs of statement allow the parser to parse ahead. For example, inserting a missing semicolon, replacing comma with a semicolon etc. Parser designers have to be careful here because one wrong correction may lead to an infinite loop.

Error Recovery

Error productions

Some common errors are known to the compiler designers that may occur in the code. In addition, the designers can create augmented grammar to be used, as productions that generate erroneous constructs when these errors are encountered.

Global correction

The parser considers the program in hand as a whole and tries to figure out what the program is intended to do and tries to find out a closest match for it, which is error-free. When an erroneous input (statement) X is fed, it creates a parse tree for some closest error-free statement Y. This may allow the parser to make minimal changes in the source code, but due to the complexity (time and space) of this strategy, it has not been implemented in practice yet.