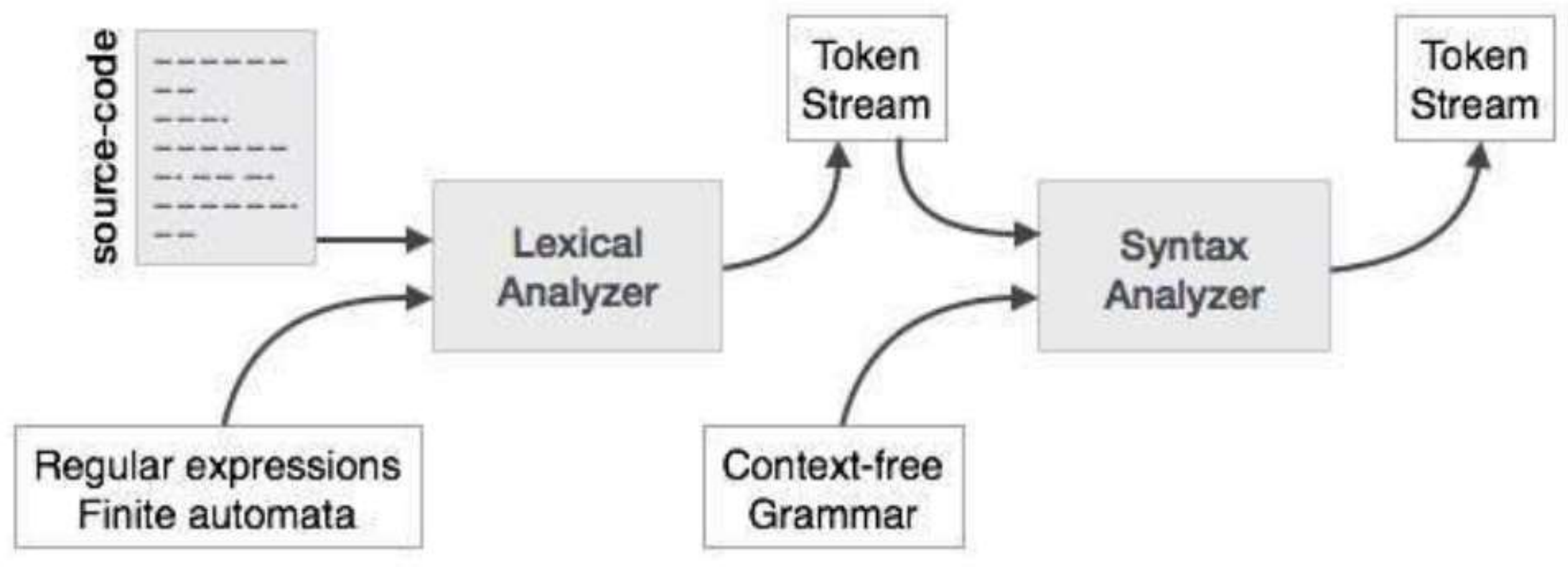
Compiler Construction

SYNTAX ANALYSIS

SYNTAX ANALYSIS



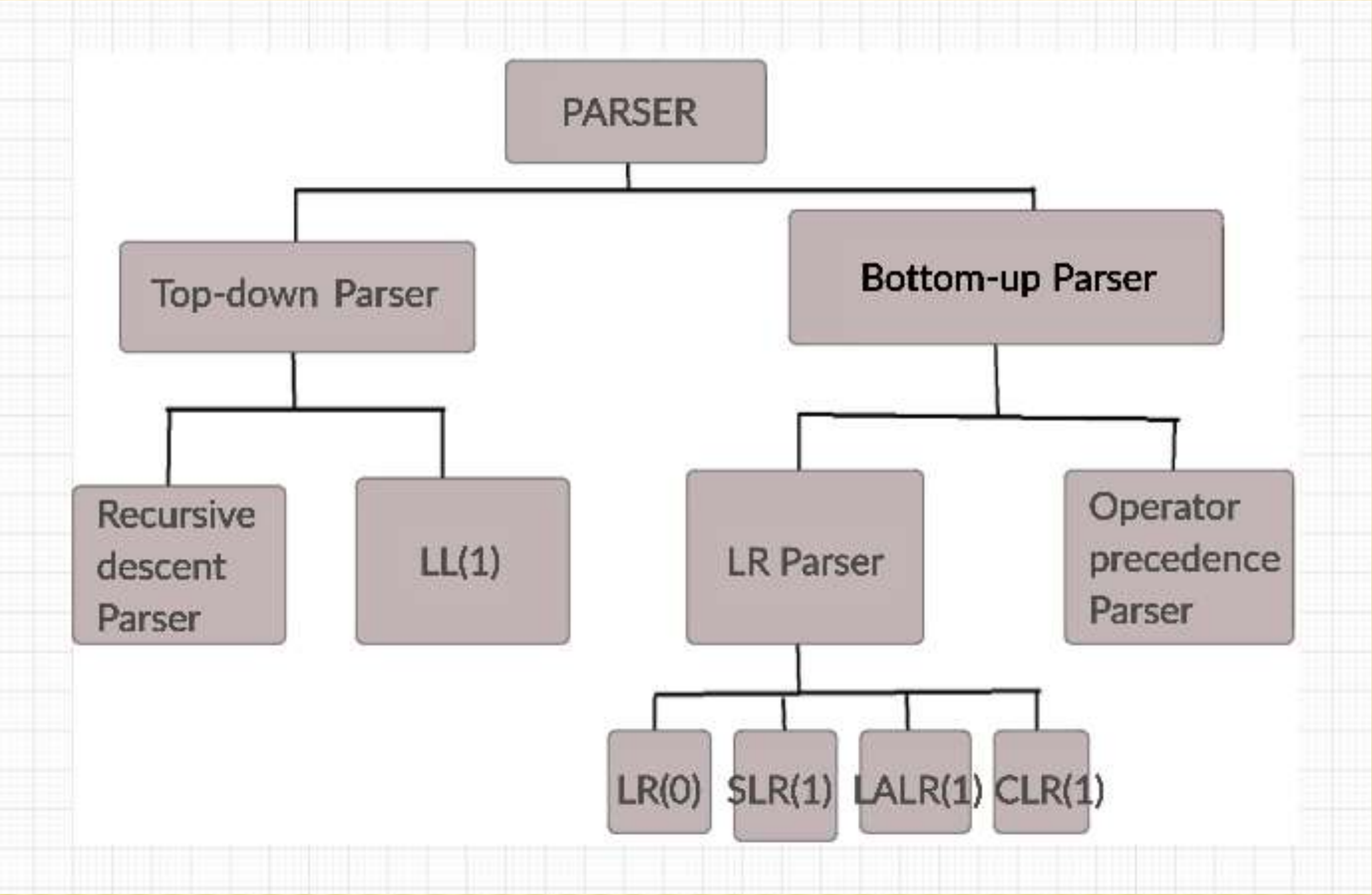
PARSING

- Parsing is the process of analyzing a text or a sequence of symbols according to the rules of a formal grammar. It is often used in computer science and linguistics to analyze and understand the structure of a sentence or a program.
- It involves breaking down the code into individual components, such as statements, functions, and variables, and checking that the syntax is correct according to the rules of the programming language.
- Overall, parsing is a fundamental process for understanding the structure and meaning of language and code.

UNIVERSAL PARSERS

- Performs parsing with any grammar.
- Hence, so called universal parsers.
- They use parsing algorithms such as Cocke-, Younger-, Kasami-algorithm or Earley's algorithm.
- This method is insufficient, so they are not used any more on commercial basis

TYPES OF PARSER

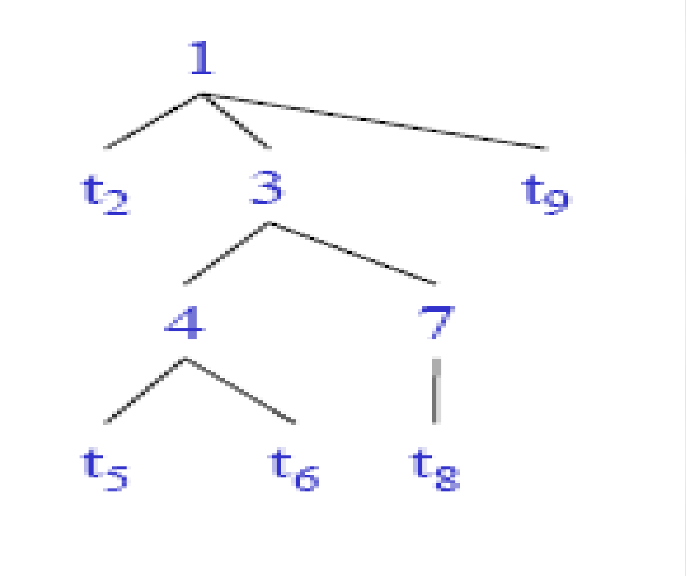


TOP DOWN PARSING

- A top-down parser is a type of parsing algorithm that starts from the highest-level, or most general, grammatical structure of a language and works its way down to the individual words and symbols.

TOP-DOWN PARSING

- The parse tree is constructed
  - From the top
  - From left to right



• Terminals are seen in order of appearance in the token stream:
t2 t5 t6 t8 t9

TOP-DOWN PARSING

- Recursive-Descent Parsing
  - Backtracking is needed (If a choice of a production rule does not work, we backtrack to try other alternatives.)
  - It is a general parsing technique, but not widely used.
  - Not efficient
- Predictive Parsing
  - no backtracking
  - efficient
  - needs a special form of grammars (LL(1) grammars).
    - Non-Recursive (Table Driven) Predictive Parser is also known as LL(1) parser.
    - Recursive Predictive Parsing is a special form of Recursive Descent parsing without backtracking.

RECURSIVE-DESCENT PARSING

- Also known as Brute Force Technique
- A non-terminal is always expanded with the first alternative only
- That is first time, first rule
- Same procedure is repeated for the newly expanded string
- This process continues until it achieves a string of terminals
- Once the nonterminal gets the string of terminals, it compares it with the input string; if it's a match, it announces successful completion
- Otherwise, it backtracks and tries the second alternative.
- If it too do not match, it backtracks to the next level and repeats the same procedure until all combinations are verified.

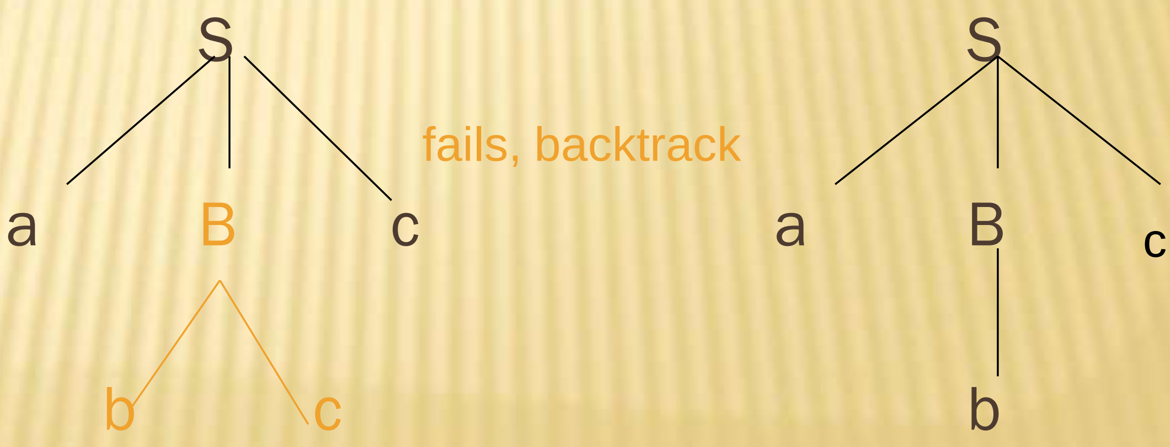
RECURSIVE-DESCENT PARSING

- Backtracking is needed.
- It tries to find the left-most derivation.

$S \rightarrow aBc$

$B \rightarrow bc \mid b$

Input : abc



RECURSIVE DESCENT PARSING

- Consider the grammar

  $E \longrightarrow T + E \;\; | \;\; T$

  $T \longrightarrow (E) \; | \;\; int \;\; | \;\; int * T$

Input: int * int

- Start with top-level non-terminal E
- Try the rules for E in order

RECURSIVE DESCENT PARSING. EXAMPLE (CONT.)

Try E ⟶ T + E

Then try a rule for T ⟶ (E)

But (does not match input token int.

Try T ⟶ int . Token matches.

But + after T does not match input token *

Try T ⟶ int * T

This will match but + after T will be unmatched

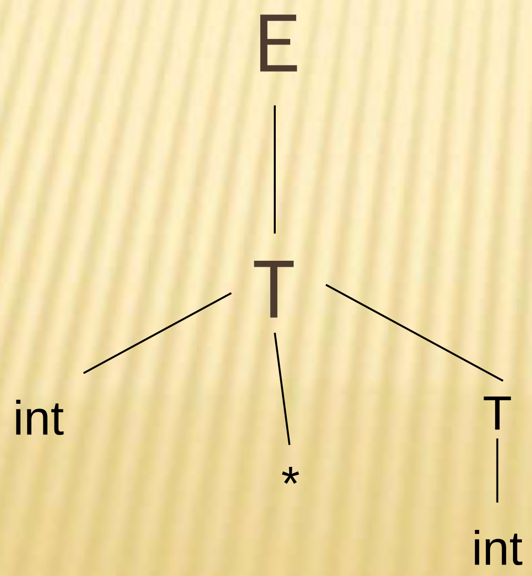
Has exhausted the choices for T

Backtrack to choose for another derivation of E

RECURSIVE DESCENT PARSING. EXAMPLE (CONT.)

Try $E \longrightarrow T$

- Follow same steps as before for T
  - – And succeed with $T \longrightarrow$ int * T and $T \longrightarrow$ int
  - – With the following parse tree



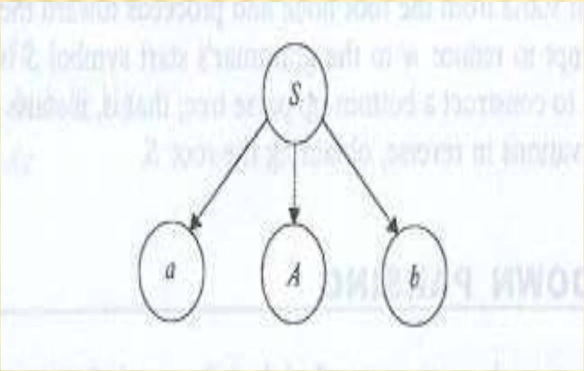
RECURSIVE-DESCENT PARSING (BACKTRACKING PROBLEM)

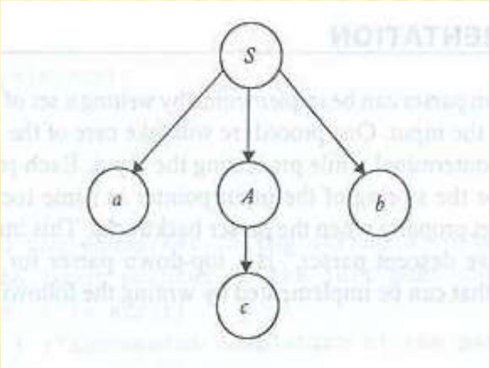
- Consider the following production

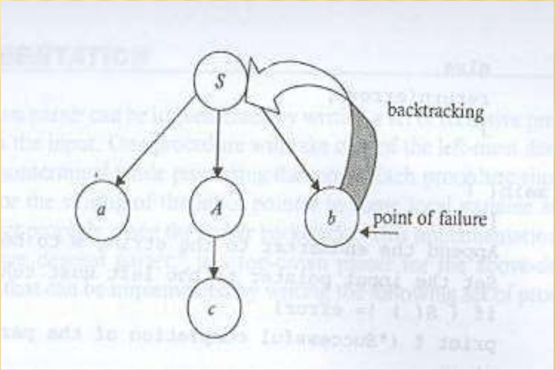
  S → aAb

  A → c |cd

Let the input string be acdb.







EXAMPLE 2

- Consider the following production

S→BA| AB

A→a| SA

B→b | SB

w= abab

Parse the above w using recursive decent parsing and find the problem of recursive decent parser