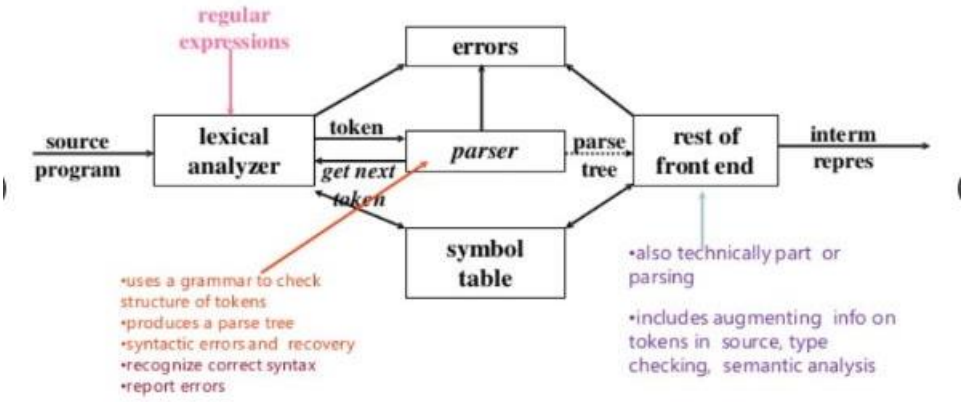
What is syntax analysis (parsing)? Explain structure of a parser.

- The process of language processing in a language processor can be divided into two primary phases: the Analysis Phase and the Synthesis Phase.
- In the Analysis Phase, the language processor takes the source program as input and performs a comprehensive analysis of it. This phase involves several subtasks, including lexical analysis, syntax analysis, and semantic analysis.
- Syntax Analysis or Parsing is the second phase, i.e. after lexical analysis. It checks the syntactical structure of the given input, i.e. whether the given input is in the correct syntax (of the language in which the input has been written) or not.
- It does so by building a data structure, called a Parse tree or Syntax tree. The parse tree is constructed by using the pre-defined Grammar of the language and the input string.
- The **syntax analyzer** (parser) basically checks for the syntax of the language. It takes a token from the lexical analyzer and group them in such a way that some programming structure (syntax) can be recognized.
- The main goal of syntax analysis is to create a parse tree or abstract syntax tree (AST) of the source code, which is a hierarchical representation of the source code that reflects the grammatical structure of the program.

Structure of the parser :



- A syntax analyzer or parser takes the input from a lexical analyzer in the form of token streams.
- The parser analyzes the source code (token stream) against the production rules to detect any errors in the code.
- This way, the parser accomplishes two tasks, i.e., parsing the code, looking for errors and generating a parse tree as the output of the phase.
- Parsers are expected to parse the whole code even if some errors exist in the program. Parsers use error recovering strategies.

Explain syntax errors & their recovery with the help of compilers.

- A parser should be able to detect and report any error in the program. It is expected that when an error is encountered, the parser should be able to handle it and carry on parsing the rest of the input.
- Planning the error handling right from the start can both simplify the structure of a compiler and improve its response to errors.
- Mostly it is expected from the parser to check for errors but errors may be encountered at various stages of the compilation process. A program may have the following kinds of errors at various stages:
  - Lexical – such as misspelling an identifier, keyword, or operator.
  - Syntax – such as an arithmetic expression with unbalanced parenthesis.
  - Semantic – such as an operator applied to an incompatible operand.
  - Logical – such as an infinitely recursive call.
- There are many different general strategies that a parser can employ to recover from a syntactic error:

a. **Panic Mode:**

- The panic mode is the simplest error recovery strategy. When an error is encountered, the parser discards the remaining input until it reaches a synchronization point, such as the end of the current statement.
- This approach does not attempt to correct the error but rather skips over it until a point where parsing can resume normally.

Advantage:

- It's easy to use.
- The program never falls into the loop.

Disadvantage:

- This technique may lead to semantic error or runtime error in further stages.

b. **Phrase Level:**

- Phrase level error recovery aims to resynchronize the parser by finding the end of the current phrase or statement.
- The parser discards input symbols until it reaches a point where it can resume parsing, typically at the end of the current statement.
- This strategy is more sophisticated than panic mode as it attempts to recover to a meaningful point in the input.

Advantages:

This method is used in many errors repairing compilers.

Disadvantages:

While doing the replacement the program should be prevented from falling into an infinite loop.

c. **Error Productions:**

- Error productions are special grammar rules added to handle syntax errors.
- When an error occurs, the parser can use these error productions to insert, delete, or replace symbols to resynchronize the parse.
- This allows the parser to continue processing the input, rather than aborting at the first error encountered.

Advantages:

1. Syntactic phase errors are generally recovered by error productions.

Disadvantages:

1. The method is very difficult to maintain because if we change the grammar then it becomes necessary to change the corresponding production.
2. It is difficult to maintain by the developers.

d. **Global Corrections:**

- Global correction strategies aim to comprehensively analyze the input and the encountered errors.
- After analysis, a series of corrections are made to the input to fix all errors at once.
- This approach requires sophisticated error analysis and correction algorithms, making it more advanced than other strategies.

Advantages:

It allows basic type conversion, which we generally do in real-life calculations.

Disadvantages:

Only Implicit type conversion is possible.

Explain following error recovery strategies:

1. **Panic Mode**
2. **Phrase Level**
3. **Error Productions**
4. **Global Corrections**

a.Panic Mode:

- The panic mode is the simplest error recovery strategy. When an error is encountered, the parser discards the remaining input until it reaches a synchronization point, such as the end of the current statement.
- This approach does not attempt to correct the error but rather skips over it until a point where parsing can resume normally.

Advantage:

- It's easy to use.
- The program never falls into the loop.

Disadvantage:

- This technique may lead to semantic error or runtime error in further stages.

b. Phrase Level:

- Phrase level error recovery aims to resynchronize the parser by finding the end of the current phrase or statement.
- The parser discards input symbols until it reaches a point where it can resume parsing, typically at the end of the current statement.
- This strategy is more sophisticated than panic mode as it attempts to recover to a meaningful point in the input.

Advantages:

This method is used in many errors repairing compilers.

Disadvantages:

While doing the replacement the program should be prevented from falling into an infinite loop.

c. Error Productions:

- Error productions are special grammar rules added to handle syntax errors.
- When an error occurs, the parser can use these error productions to insert, delete, or replace symbols to resynchronize the parse.

- This allows the parser to continue processing the input, rather than aborting at the first error encountered.

Advantages:

2. Syntactic phase errors are generally recovered by error productions.

Disadvantages:

3. The method is very difficult to maintain because if we change the grammar then it becomes necessary to change the corresponding production.
4. It is difficult to maintain by the developers.

d. Global Corrections:

- Global correction strategies aim to comprehensively analyze the input and the encountered errors.
- After analysis, a series of corrections are made to the input to fix all errors at once.
- This approach requires sophisticated error analysis and correction algorithms, making it more advanced than other strategies.

Advantages:

It allows basic type conversion, which we generally do in real-life calculations.

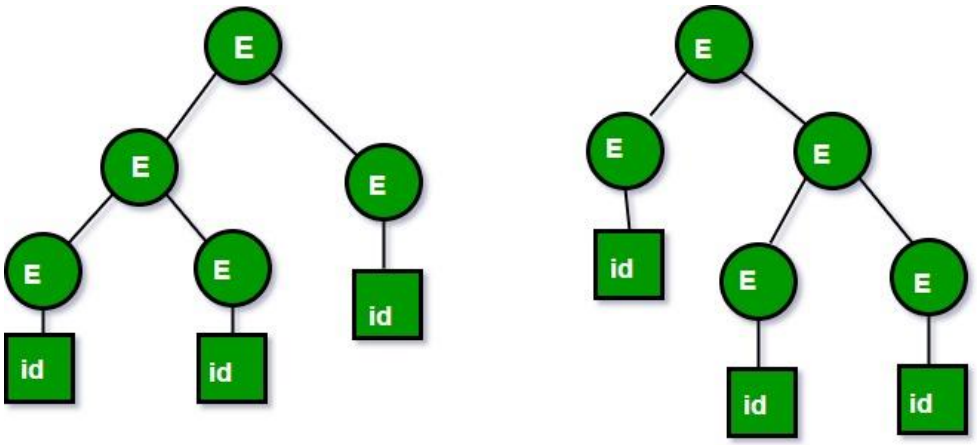
Disadvantages:

Only Implicit type conversion is possible.

What is ambiguity in grammar? Explain with example.

- A Context-Free Grammar (CFG) is considered ambiguous if there exists more than one derivation tree for a given input string. This means there are multiple LeftMost Derivation Trees (LMDT) or RightMost Derivation Trees (RMDT) for the same input string.

Example 1: Consider the grammar:



2. Let us now consider the following grammar:

Set of alphabets ? = {0,…,9, +, *, (,)}

E -> I

E -> E + E

E -> E * E

E -> (E)

I -> ? | 0 | 1 | … | 9

From the above grammar String **3*2+5** can be derived in 2 ways:

I) First leftmost derivation

E=>E*E

=>I*E

=>3*E+E

=>3*I+E

=>3*2+E

=>3*2+I

=>3*2+5

II) Second leftmost derivation

E=>E+E

=>E*E+E

=>I*E+E

=>3*E+E

=>3*I+E

=>3*2+I

=>3*2+5

Both the above parse trees are derived from the same grammar rules but both parse trees are different. Hence the grammar is ambiguous

Inherently Ambiguous Language:

- An inherently ambiguous language is one where every CFG representing the language is ambiguous. This means there is no way to rewrite the grammar to remove ambiguity.

Questions on Ambiguous Grammar:

1. **Determining Ambiguity:**
    - If a grammar exhibits both left and right recursion, it is likely ambiguous. For example, **S -> SaS | ?** is ambiguous because it contains both left and right recursion.
2. **Unambiguity despite no Left/Right Recursion:**
    - The absence of left or right recursion in a grammar does not guarantee unambiguity. For instance, **S -> aB | ab**, **A -> AB | a**, **B -> Abb | b** is ambiguous despite lacking left or right recursion.
3. **Identifying Ambiguity:**
    - For a grammar like **S -> SAB | ?**, we can observe that by replacing **B -> AS** in **S -> SAB**, we get **S -> SAAS**, showing both left and right recursion. Thus, the grammar is ambiguous.

Ambiguity in grammar can lead to multiple interpretations of the same input string, which is undesirable in programming languages as it can result in unexpected behavior. Identifying and resolving ambiguity is crucial for ensuring the correctness and predictability of language parsing.

What is precedence & associativity of operators? Explain with example

- An ambiguous grammar may be converted into an unambiguous grammar by implementing the precedence and Associativity constraints.

Precedence

Precedence refers to the order in which operators are evaluated in an expression. Operators with higher precedence are evaluated before those with lower precedence. For example, in the expression 3 + 4 * 5, the multiplication operator * has higher precedence than the addition operator +, so 4 * 5 is evaluated first, resulting in 20, and then 3 + 20 is evaluated to give 23.

Associativity

Associativity determines the order in which operators of the same precedence are evaluated. Operators can be left-associative, right-associative, or non-associative.

- Left-associative: Operators are evaluated from left to right. For example, in a - b - c, the subtraction operator is left-associative, so a - b is evaluated first, and then the result is subtracted from c.
- Right-associative: Operators are evaluated from right to left. For example, the exponentiation operator ^ is right-associative, so a ^ b ^ c is evaluated as a ^ (b ^ c).
- Non-associative: Operators do not associate in any direction. For example, the comparison operator == is non-associative, so expressions like a == b == c are not valid.

These constraints are implemented using the following rules –

Rule-01:

The precedence of operators is implemented using following rules-

- The level at which the production is present defines the priority of the operator contained in it.
- The higher the level of the production, the lower the priority of operator.
- The lower the level of the production, the higher the priority of operator.

Rule-02:

- The Associativity of operators is implemented using following rules
- If the operator is left associative, induce left recursion in its production.
- If the operator is right associative, induce right recursion in its production.

Example

Consider the expression 3 + 4 * 5 / 2. The operators in this expression have different precedence levels:

1. Multiplication * and division / have higher precedence than addition +.

2. Multiplication and division have the same precedence and are left-associative.
3. Addition has lower precedence than multiplication and division.

Following the precedence and associativity rules:

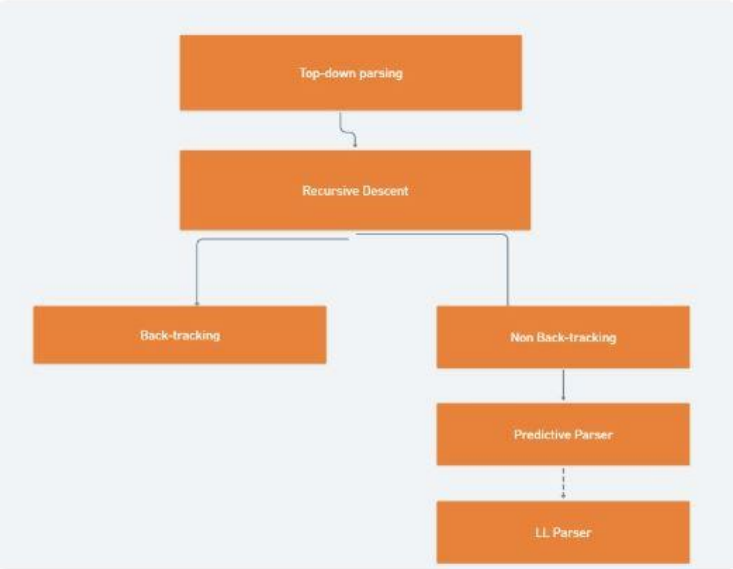
1. 4 * 5 is evaluated first, resulting in 20.
2. 20 / 2 is evaluated next, resulting in 10.
3. Finally, 3 + 10 is evaluated, giving the final result of 13.

What is top down parsing? Explain with example.

- Top-down parsing is a parsing technique where the parser starts with the start symbol of the grammar and attempts to transform it into the input string by repeatedly applying production rules.
- In top-down parsing, the parse tree is generated from top to bottom, i.e., from root to leaves & expand till all leaves are generated.
- It follows a leftmost derivation strategy, meaning it expands the leftmost nonterminal in the current sentential form during each step of parsing.

Types of Top-Down Parsing

- There are two types of top-down parsing which are as follows –



Working of Top-Down Parsing:

- Consider the grammar:

S -> aABe

A -> Abc | b

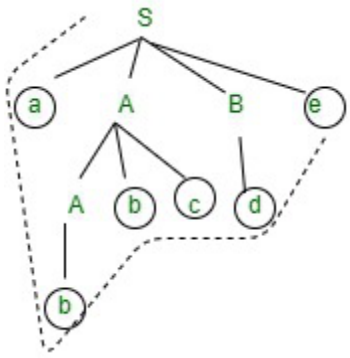
B -> d

Input – abbcde$

- First, you can start with S -> a A B e and then you will see input string a in the beginning and e in the end.
- Now, you need to generate abbcde .
- Expand A-> Abc and Expand B-> d.
- Now, You have string like aAbcde and your input string is abbcde.

- Expand A->b.

- Final string, you will get **abbcde**.
- The input string has been completely consumed, and the parse tree is successfully constructed.

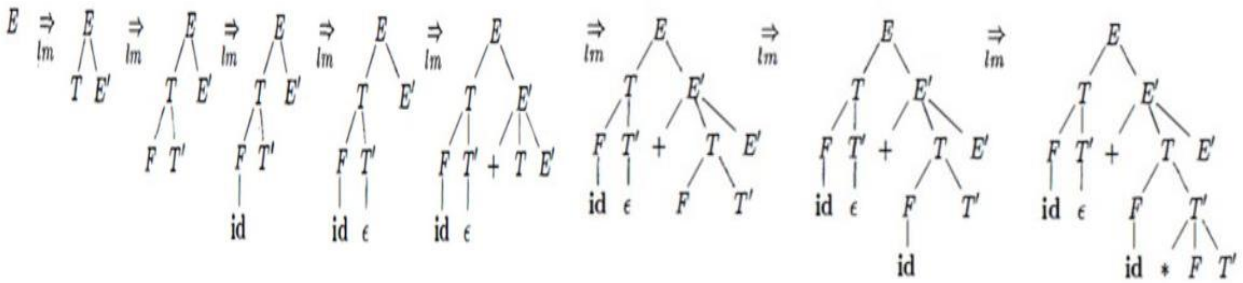


Example2 –

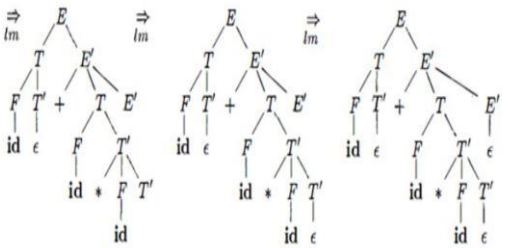
Top Down Parsing for id+id*id

$$
\begin{aligned}
E &\rightarrow T\ E' \\
E' &\rightarrow +\ T\ E' \mid \epsilon \\
T &\rightarrow F\ T' \\
T' &\rightarrow *\ F\ T' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

Top Down Parsing for id+id*id



Top Down Parsing for id+id*id



What are recursive descent parsers? Explain with example.

- Top-down parsing is a parsing technique where the parser starts with the start symbol of the grammar and attempts to transform it into the input string by repeatedly applying production rules.
- Recursive Descent Parser uses the technique of Top-Down Parsing with backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with backtracking.
- It can be simply performed using a Recursive language. The first symbol of the string of R.H.S of production will uniquely determine the correct alternative to choose.

How Recursive Descent Parser Works –

1. **Start at the Top Level**: The RDP begins by identifying the starting rule of the grammar, typically the top-level non-terminal symbol. This symbol represents the highest-level construct in the language, such as a program or a statement.
2. **Call Parsing Function**: The parser calls the corresponding parsing function for the identified non-terminal symbol. This parsing function is responsible for analyzing the syntax of the construct represented by the non-terminal symbol.
3. **Recursive Descent**: Within the parsing function, the parser recursively descends through the grammar rules associated with the non-terminal symbol. It does this by calling parsing functions for each sub-rule or alternative rule.
4. **Token Comparison**: At each step of the recursion, the parser compares the current input token to the expected token(s) according to the grammar rule being processed. It uses lookahead to determine which rule to follow next based on the current input token.
5. **Token Consumption**: If the current input token matches the expected token, the parser consumes the token and moves on to the next one. If the input token does not match the expected token, the parser generates a syntax error and halts further processing.
6. **Building the Parse Tree**: As the parser descends through the grammar rules, it constructs a parse tree that represents the syntactic structure of the input text according to the grammar rules of the language.
7. **Error Handling**: If a syntax error is encountered during the parsing process, the parser may attempt error recovery strategies such as panic mode or statement mode to continue parsing and provide meaningful error messages to the user.
8. **Semantic Analysis**: Once the parsing process is complete and no syntax errors are found, the parse tree can be used for further analysis, such as semantic analysis or code generation, depending on the requirements of the compiler or interpreter.

Example1 – Consider the Grammar

S → a A d

A → b c | b

Make parse tree for the string a bd. Also, show parse Tree when Backtracking is required when the wrong alternative is chosen.

Solution

The derivation for the string **abd** will be –

S ⇒ a A d ⇒ abd (Required String)

If bc is substituted in place of non-terminal A then the string obtained will be abcd.

S ⇒ a A d ⇒ abcd (Wrong Alternative)

Figure (a) represents S → aAd

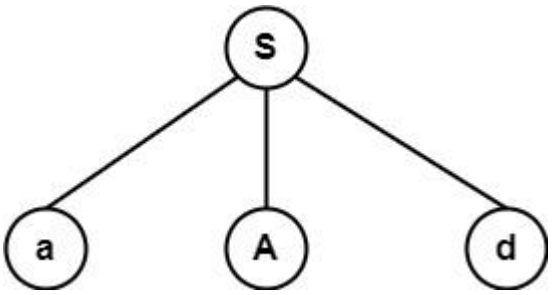
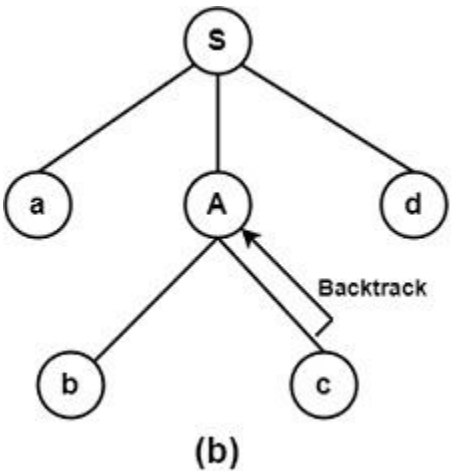
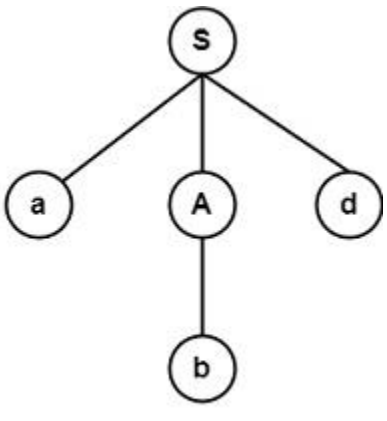


Figure (b) represents an expansion of tree with production A → bc which gives string abcd which does not match with string abd.

So, it backtracks & chooses another alternative, i.e., A → b in figure (c) which matches with abd.



(b)



(c)

What is backtracking? Explain with example.

- Backtracking is a technique used in parsing algorithms, such as Recursive Descent Parsing, to handle situations where the parser makes a wrong choice and needs to backtrack or reconsider previous decisions to find the correct parsing path.
- It involves undoing or retracting choices made during parsing and trying alternative options.
- Top-down parsing is a parsing technique where the parser starts with the start symbol of the grammar and attempts to transform it into the input string by repeatedly applying production rules.
- Recursive descent is a top-down parsing technique that constructs the parse tree from the top and the input is read from left to right.
- Recursive Descent Parser uses the technique of Top-Down Parsing with backtracking. It can be defined as a Parser that uses the various recursive procedure to process the input string with backtracking.
- Example of Recursive Descent using backtracking –

Example1 – Consider the Grammar

S → a A d

A → b c | b

Make parse tree for the string a bd. Also, show parse Tree when Backtracking is required when the wrong alternative is chosen.

Solution

The derivation for the string **abd** will be –

S ⇒ a A d ⇒ abd (Required String)

If bc is substituted in place of non-terminal A then the string obtained will be abcd.

S ⇒ a A d ⇒ abcd (Wrong Alternative)

Figure (a) represents S → aAd

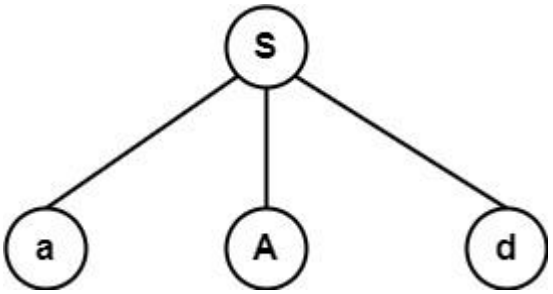
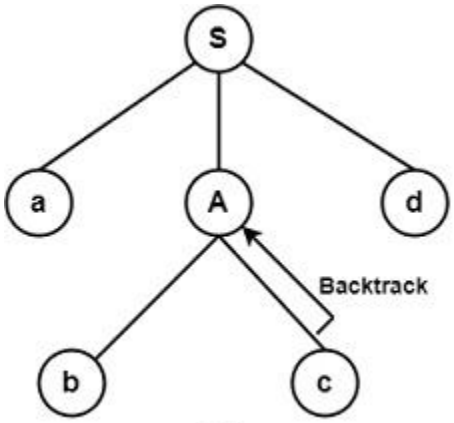
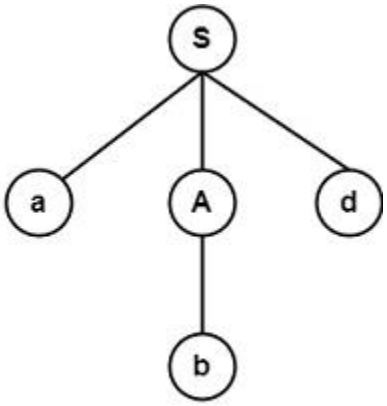


Figure (b) represents an expansion of tree with production A → bc which gives string abcd which does not match with string abd.

So, it backtracks & chooses another alternative, i.e., A → b in figure (c) which matches with abd.



(b)



(c)

How to design non backtracking recursive descent parsers?

- Top-down parsing is a parsing technique where the parser starts with the start symbol of the grammar and attempts to transform it into the input string by repeatedly applying production rules.
- There are two types of Top-Down Parsers:
  - Top-Down Parser with Backtracking
  - Top-Down Parsers without Backtracking
- Non-recursive descent parser is also known as LL(1) parser or predictive parser or without backtracking parser or dynamic parser. It uses a parsing table to generate the parse tree instead of backtracking.

LL(1) Parsing:

- A top-down parser that uses a one-token lookahead is called an LL(1) parser.
- Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree.
- Finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

Given below is an algorithm for LL(1) Parsing:

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 */
```

Example 1: Consider the Grammar:

E --> TE'
E' --> +TE' | ε
T --> FT'
T' --> *FT' | ε
F --> id | (E)

*ε denotes epsilon

Step 1: The grammar satisfies all properties in step 1.

Step 2: Calculate first() and follow().

Find their First and Follow sets:

	First	Follow
E –> TE'	{ id, (}	{ $,) }
E' –> +TE'/ ε	{ +, ε }	{ $,) }
T –> FT'	{ id, (}	{ +, $,) }
T' –> *FT'/ ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Step 3: Make a parser table.

Now, the LL(1) Parsing Table is:

	id	+	*	(	)	$
E	E –> TE'			E –> TE'		
E'		E' –> +TE'			E' –> ε	E' –> ε
T	T –> FT'			T –> FT'		
T'		T' –> ε	T' –> *FT'		T' –> ε	T' –> ε
F	F –> id			F –> (E)		

As you can see that all the null productions are put under the Follow set of that symbol and all the remaining productions lie under the First of that symbol.

Explain non recursive predictive parsing algorithm.
. Explain LL (1) parsing algorithm.

- Predictive parsing is a special form of recursive descent parsing, where no backtracking is required, so this can predict which products to use to replace the input string.
- Non-recursive predictive parsing or table-driven is also known as LL(1) parser. This parser follows the leftmost derivation (LMD).

LL(1) Parsing:

- A top-down parser that uses a one-token lookahead is called an LL(1) parser.
- Here the 1st **L** represents that the scanning of the Input will be done from the Left to Right manner and the second **L** shows that in this parsing technique, we are going to use the Left most Derivation Tree.
- Finally, the **1** represents the number of look-ahead, which means how many symbols are you going to see when you want to make a decision.

- The main problem during predictive parsing is that of determining the production to be applied for a non-terminal.
- This non-recursive parser looks up which product to be applied in a parsing table. A LL(1) parser has the following components:

(1) buffer: an input buffer which contains the string to be passed
(2) stack: a pushdown stack which contains a sequence of grammar symbols
(3) A parsing table: a 2d array M[A, a]
where
A->non-terminal, a->terminal or $
(4) output stream:
end of the stack and an end of the input symbols are both denoted with $

Given below is an algorithm for LL(1) Parsing:

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 $X \in V_t$ 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 */
```

breakdown of the algorithm:

1. **Input**: The algorithm takes as input a string ω to be parsed and a parsing table M for the grammar G.
2. **Output**: If the input string ω belongs to the language generated by the grammar G (i.e., ω is in L(G)), the algorithm outputs the left-most derivation of ω. Otherwise, it outputs an error.
3. **Initialization**: The algorithm starts with the initial state where the start symbol $S is on the stack and the input string ω$ is in the input buffer. The input pointer (ip) is set to point to the first symbol of ω$.
4. **Parsing Loop**: The algorithm repeats the following steps until the stack contains only $ (empty stack):

a. **Read Symbols**: It reads the top symbol X from the stack and the symbol a pointed by the input pointer ip.

b. **Terminal Check**: If X is a terminal symbol or $, it checks whether X matches the symbol a.

- If X = a, it pops X from the stack and advances the input pointer ip.
- If X ≠ a, it outputs an error.

c. **Non-terminal Expansion**: If X is a non-terminal symbol:

- It consults the parsing table M to find the production rule for X and the lookahead symbol a.
- If M[X,a] contains a production X → Y1, Y2,... Yk, it pops X from the stack and pushes Y1, Y2,... Yk onto the stack in reverse order (Y1 on top).
- It outputs the production X → Y1, Y2,... Yk.

d. **Error Handling**: If no applicable production rule is found in the parsing table M for the current stack symbol X and lookahead symbol a, it outputs an error.

5. **Termination**: The parsing loop terminates when the stack contains only $, indicating that the parsing process is complete.

What is operator precedence parsing? Explain with example.

- A grammar that is used to define mathematical operators is called an operator grammar or operator precedence grammar.
- Such grammars have the restriction that no production has either an empty right-hand side (null productions) or two adjacent non-terminals in its right-hand side.
- Operator precedence parsing is a type of Shift Reduce Parsing. In operator precedence parsing, an operator grammar and an input string are fed as input to the operator precedence parser, which may generate a parse tree.
- In operator precedence parsing, the shift and reduce operations are done based on the priority between the symbol at the top of the stack and the current input symbol.
- The operator precedence parser performs the operator precedence parsing using operator grammar.
- A grammar is said to be an operator grammar if it follows these two properties:
  - There should be no ε (epsilon) on the right-hand side of any production.
  - There should be no two non-terminals adjacent to each other.
  - Examples of operator grammar are A + B, A - B, A x B, etc.
- There are three operator precedence relations. These are-
  - a ⋗ b: This relation implies that terminal “a” has higher precedence than terminal “b”.
  - a ⋖ b: This relation implies that terminal “a” has lower precedence than terminal “b”.
  - a ≐ b: This relation implies that terminal “a” has equal precedence to terminal “b”.
- A precedence table is used in operator precedence parsing to establish the relative precedence of operators and to resolve shift-reduce conflicts during the parsing process.

Parsing Action :

- Both end of the given input string, add the $ symbol.
- Now scan the input string from left right until the ⋗ is encountered.
- Scan towards left over all the equal precedence until the first left most ⋖ is encountered.
- Everything between left most ⋖ and right most ⋗ is a handle.
- $ on $ means parsing is successful.

Example

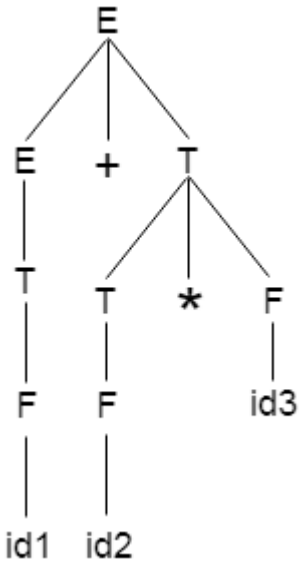
Grammar:

1. E → E+T/T
2. T → T*F/F
3. F → id

Given string:

1. w = id + id * id

Let us consider a parse tree for it as follows:



On the basis of above tree, we can design following operator precedence table:

	E	T	F	id	+	*	$
E	X	X	X	X	≐	X	⋗
T	X	X	X	X	⋗	≐	⋗
F	X	X	X	X	⋗	⋗	⋗
id	X	X	X	X	⋗	⋗	⋗
+	X	≐	⋖	⋖	X	X	X
*	X	X	≐	⋖	X	X	X
$	⋖	⋖	⋖	⋖	X	X	X

Now let us process the string with the help of the above precedence table:

$ ⋖ id1 ⋗ + id2 * id3 $

$ ⋖ F ⋗ + id2 * id3 $

$ ⋖ T ⋗ + id2 * id3 $

$ ⋖ E ≐ + ⋖ id2 ⋗ * id3 $

$ ⋖ E ≐ + ⋖ F ⋗ * id3 $

$ ⋖ E ≐ + ⋖ T ≐ * ⋖ id3 ⋗ $

$ ⋖ E ≐ + ⋖ T ≐ * ≐ F ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ≐ + ≐ T ⋗ $

$ ⋖ E ⋗ $

Accept.

Explain SLR parsing.

Reference articles to read –

https://ebooks.inflibnet.ac.in/csp10/chapter/slr-parsing/

https://www.naukri.com/code360/library/slr-parser-in-compiler-design

https://www.javatpoint.com/slr-1-parsing

https://www.geeksforgeeks.org/slr-parser-with-examples/

Explain LALR parsing

Reference articles to read –

https://www.naukri.com/code360/library/lalr-parser-in-compiler-design

https://www.tutorialspoint.com/what-is-lalr-1-parser

https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/140%20LALR%20Parsing.pdf

https://www.geeksforgeeks.org/lalr-parser-with-examples/

How to calculate FIRST & FOLLOW sets?

Reference articles to read –

https://www.slideshare.net/slideshow/ll1-parsing/168595061