Top-Down Parsing

Top-down parsing constructs parse tree for the input string, starting from root node and creating the nodes of parse tree in pre-order.

It is done by leftmost derivation for an input string.

$$E \longrightarrow E+E \mid E*E \mid id$$

Solution:

$w = id + id * id$

$E \xrightarrow[lm]{} E+E$

$E \xrightarrow[lm]{} id+E \quad [E \longrightarrow id]$

$E \xrightarrow[lm]{} id+E*E \quad [E \longrightarrow E*E]$

$E \xrightarrow[lm]{} id+id*E \quad [E \longrightarrow id]$

$E \xrightarrow[lm]{} id+id*id \quad [E \longrightarrow id]$

General Strategies

• Top-down parsing involves constructing the parse tree starting from root node to leaf node by consuming tokens generated by lexical analyzer.

• Top-down parsing is characterized by the following methods:

• ***Brute-force method,*** accompanied by a parsing algorithm.

  o All possible combinations are attempted before the failure to parse is recognized.

• ***Recursive descent,*** is a parsing technique which does not allow backup.

  o Involves backtracking and left recursion.

• ***Top-down parsing*** with limited or partial backup.

Recursive Descent Parser

• Recursive descent parser is a top-down parser.

• It requires backtracking to find the correct production to be applied.

• The parsing program consists of a set of procedures, one for each non-terminal.

• Process begins with the procedure for start symbol.

• Start symbol is placed at the root node and on encountering each non-terminal, the procedure concerned is called to expand the non-terminal with its corresponding production.

• Procedure is called recursively until all non-terminals are expanded.

• Successful completion occurs when the scan over entire input string is done. ie., all terminals in the sentence are derived by parse tree.

```
void A()
{
      choose an A-production, A ----> X1 X2 X3... Xk;
        for (i = 1 to k)
            if (Xi is a non-terminal)
              call procedure Xi ();
              else if (Xi equals the current input symbol a)
                    advance the input to the next symbol;
              else
                    error;
}
```

Limitation

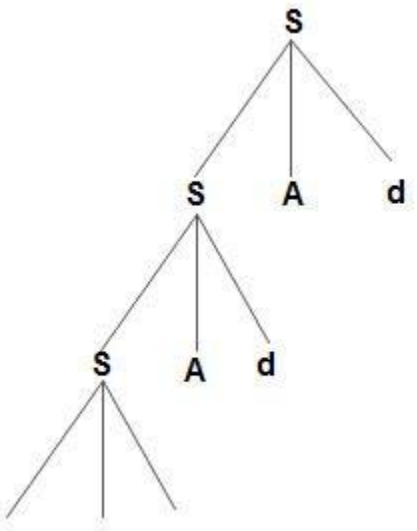
• When a grammar with left recursive production is given, then the parser might get into infinite loop.

Hence, left recursion must be eliminated.

(eg.) Let grammar G be,

S ----> SAd

A ---> ab I d



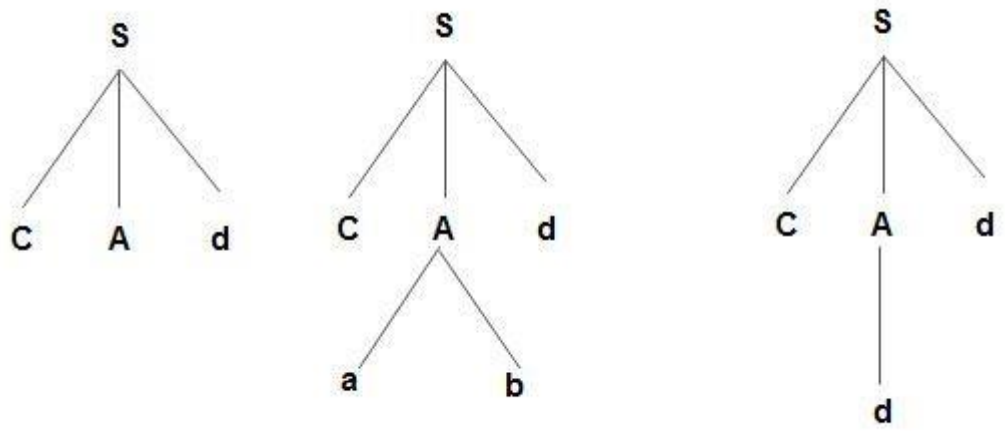
Recursive descent parser with backtracking

(eg.) Let grammar G be,

S ----> cAd

A ----> ab | d

w = cad



Explanation

• The root node contains the start symbol which is S.

• The body of production begins with c, which matches with the first symbol of the input string.

• A is a non-terminal which is having two productions A ----> *ab* I *d.*

• Apply the first production of A, which results in the string *cabd* that does not match with the given string *cad.*

• Backtrack to the previous step where the production of A gets expanded and try with alternate production of it.

• This produces the string *cad* that matches with the given string.

Limitation

• If the given grammar has more number of alternatives then the cost of backtracking will be high.

Recursive descent parser without backtracking

Recursive descent parser without backtracking works in a similar way as that of recursive descent parser with backtracking with the difference that each non-terminal should be expanded by its correct alternative in the first selection itself.

When the correct alternative is not chosen, the parser cannot backtrack and results in syntactic error.

Advantage

• Overhead associated with backtracking is eliminated.

Limitation

• When more than one alternative with common prefixes occur, then the selection of the correct alternative is highly difficult.

Hence, this process requires a grammar with no common prefixes for alternatives.

Predictive Parser / LL(1) Parser

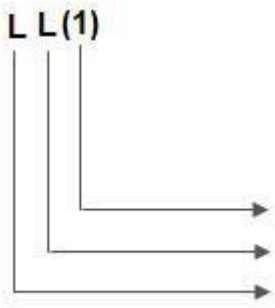
• Predictive parsers are top-down parsers.

• It is a type of recursive descent parser but with *no backtracking.*

• It can be implemented non-recursively by using stack data structure.

• They can also be termed as LL (l) parser as it is constructed for a class of grammars called LL (l).

• The production to be applied for a non-terminal is decided based on the current input symbol.



A grammar G is LL(l) if there are two distinct productions A ---> α | βwith the following conditions hold:

o For no terminal α and βderive strings beginning with a.

o At most one of α and βcan derive empty string.

o If β *---> Ɛ then α does not derive any string beginning with a terminal in FOLLOW(A).

o If α *---> Ɛ then βdoes not derive any string beginning with a terminal in FOLLOW(A).

In order to overcome the limitations of recursive descent parser, **LL(1)** parser is designed by using stack data structure explicitly to hold grammar symbols.

In addition to this,

• Left recursion is eliminated.

• Common prefixes are also eliminated (Left factoring).

Eliminating left recursion

A grammar is left recursive if it has a production of the form A ----> A α, for some string α.

To eliminate left recursion for the production, A ---> A α I β

Rule

A ---> β *A′*

A' ---> αA' I Ɛ

Example

A ----> A α_1 | A α_2 | · · · | β_1 | β_2 | · · · | β_m

Solution:

A----> β_1A' | β_2A' | ⋯ | β_mA'

A'----> α_1A' | α_2A'

Left factoring

When a production has more than one alternatives with common prefixes, then it is necessary to make right choice on production.

This can be done through rewriting the production until enough of the input has been seen.

To perform left-factoring for the production, A ---> $\alpha\beta_1$ | $\alpha\beta_2$

Rule

A ---> α *A′*

A' ---> β_1I β_2

Example

A -> α β_1 I α β_2 I ⋯I α β_m I ɣ

Solution

A-> α A'

A' -> β1I β2I ⋯ I βm

Computation of FIRST

FIRST(α) is the set of terminals that begin strings derived from α.

Rules

• To compute FIRST(X), where X is a grammar symbol,

• If X is a terminal, then FIRST(X) = {X}.

• If X -> Ɛis a production, then add Ɛto FIRST (X).

• If X is a non-terminal and X-> $Y_1Y_2 \cdot \cdot \cdot Y_k$is a production, then add FIRST(Y_1) to FIRST(X). If Y_1 derives c, then add FIRST(Y_2) to FIRST(X).

Computation of FOLLOW

FOLLOW(A) is the set of terminals *a,* that appear immediately to the right of A.

For rightmost sentential form of A, $ will be in FOLLOW(A).

Rules

• For the FOLLOW(Start symbol) place $, where $ is the input end marker.

• If there is a production A -> α *B* β, then everything in FIRST(β) except Ɛis in FOLLOW(A).

• If there is a production A -> α B, or a production A -> α *B* β where FIRST((β) contains Ɛ, then everything in FOLLOW(A) is in FOLLOW(B).

Construction of parsing table

Algorithm Construction of predictive parsing table

`Input` Grammar G

`Output` Parsing table M

`Method` For each production `A --> α`, do the following:

1. For each terminal αin FIRST(α), add A `-->`αto M[A, a].

2. If Ɛis in FIRST (α) , then for each terminal *b* in FOLLOW(A) ' add A `-->` αto M[A, b]

3. If Ɛ is in FIRST (Ɛ) and $ is in FOLLOW(A) , add A `-->` αto M[A, $] .

4. If no production is found in M[A, a] then set error to M[A, a].

Note:

In general, parsing table entry will be empty for indicating error status.

Parsing of input

Predictive parser contains the following components:

• Stack - holds sequence of grammar symbols with $ on the bottom of stack

• Input buffer - contains the input to be parsed with $ as an end marker for the string.

• Parsing table.

Process

• Initially the stack contains $ to indicate bottom of the stack and the start symbol of grammar on top of $.

• The input string is placed in input buffer with $ at the end to indicate the end of the string.

• Parsing algorithm refers the grammar symbol on the top of stack and input symbol pointed by the pointer and consults the entry in *M[A,* α] where *A* is in top of stack and αis the symbol read by the pointer.

• Based on the table entry, if a production is found then the tail of the production is pushed onto stack in reversal order with leftmost symbol on the top of stack.

• Process repeats until the entire string is processed.

• When the stack contains $ (bottom end marker) and the pointer reads $ (end of input string), successful parsing occurs.

• If no entry is found, it reports error stating that the input string cannot be parsed by the grammar.

Algorithm Table-driven predictive parsing

Input A string w and parsing table M for a grammar G

Output If w is in L(G) then success; otherwise error

Method

```
Let a be the first symbol of w;
Let X be the top of stack symbol;
while(X ≠ $)
if (X = a) pop the stack and let a be the next
symbol of w;
else if (X is a terminal) error();
else if (M[X, a] is an error entry) error();
else if (M[X, a]=X-->Y1Y2...Yk){
output the production X-->Y1Y2...Yk;
pop the stack;
push Yk,Yk-1 r ... ,Y1 onto
stack with Y1 on the top;}
Let X be the top stack symbol; }
```

Non-recursive Predictive Parser

Non-recursive predictive parser uses explicit stack data structure.

This prevents implicit recursive calls.

It can also be termed as table-driven predictive parser.

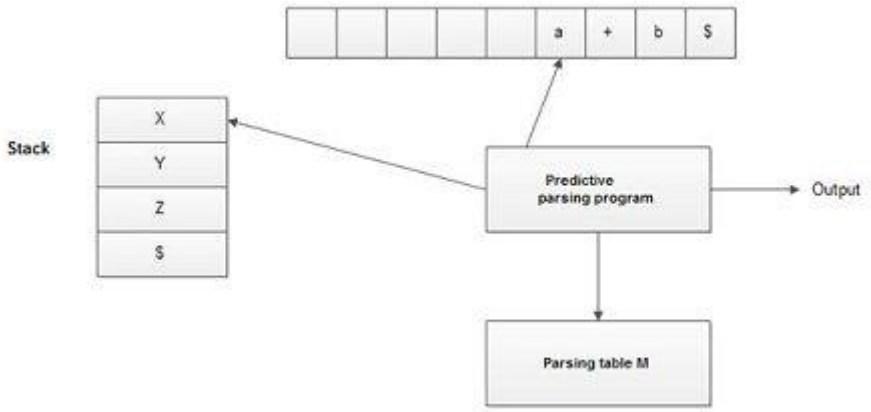
Components

• Input buffer - holds input string to be parsed.

• Stack - holds sequence of grammar symbols.

• Predictive parsing algorithm - contains steps to parse the input string; controls the parser's process.

• Parsing table - contains entries based on which parsing action has to be carried out.



Process

• Initially, the stack contains $ at the bottom of the stack.

• The input string to be parsed is placed in the input buffer with $ as the end marker.

• If X is a non-terminal on the top of stack and the input symbol being read is *a,* the parser chooses a production by consulting entry in the parsing table M[X, *a].*

• Replace the non-terminal in stack with the production found in M[X, *a]* in such a way that the leftmost symbol of right side of production is on the top of stack i.e., the production has to be pushed to stack in reverse order.

• Compare the top of stack symbol with input symbol.

• If it matches, pop the symbol from stack and advance the pointer reading the input buffer.

• If no match is found repeat from step 2.

• Stop parsing when the stack is empty (holds $) and input buffer reads end marker ($).

Error Recovery in Predictive Parsing

• Recovery in a non-recursive predictive parser is easier than in a recursive descent parser.

• Panic mode recovery

  o If a terminal on stack, pop the terminal.

  o If a non-terminal on stack, shift the input until the terminal can expand.

• Phrase level recovery

o Carefully filling in the blank entries about what to do.

BOTTOM-UP PARSING

• Bottom-up parsers construct parse trees starting from the *leaves* and work up to the *root.*

• Bottom-up syntax analysis is also termed as shift-reduce parsing.

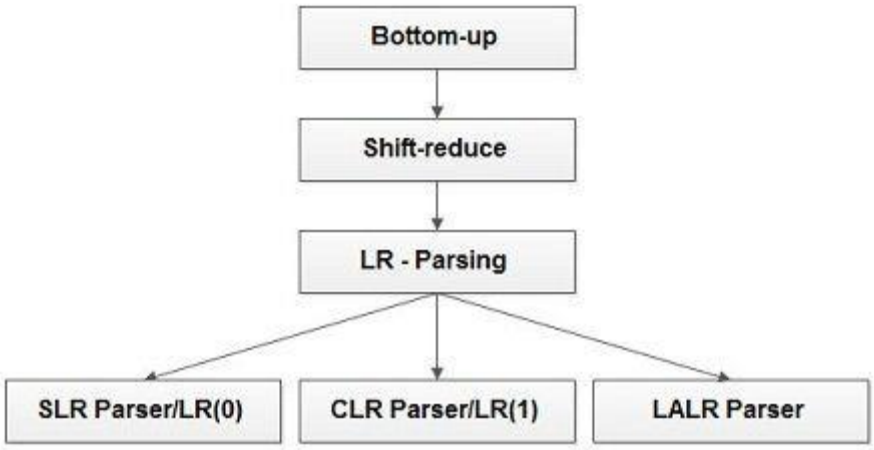
• The common method of shift-reduce parsing is called LR parsing.

• Operator precedence parsing is an easy-to-implement shift-reduce parser.

• Shift-reduce parsing try to build a parse tree for an input string beginning at the leaves (the bottom) and working up towards the root (the top).

• At each and every step of reduction, the right side of a production which matches with the substring is replaced by the left side symbol of the production.

• If the substring is chosen correctly at each step, a rightmost derivation is traced out in reverse.

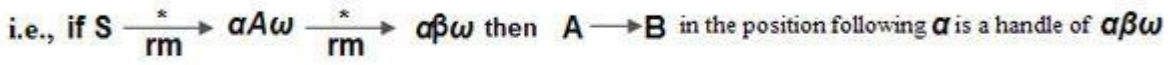


Handles

A *handle* of a string is a substring that matches the right side of a production and whose reduction to the non-terminal on the left side of the production represents one step along the reverse of a rightmost derivation.

Precise definition of a handle

• A handle of a right-sentential form ɣ is a production A −−>βand a position *of* ɣ where the string βmay be found and replaced by A to produce the previous right-sentential form in a rightmost derivation *of* ɣ*.*



• The string *w* to the right of the handle contains only terminal symbols.

(eg.) Consider the grammar

S −−>*αABe*

A −−> *Abc* I *b*

B −−> *d*

The sentence *abbcde* can be reduced to S by the following steps.

abbcde

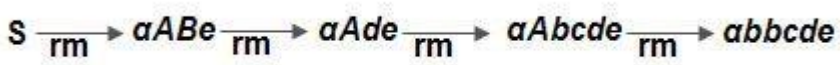
aAbcde

aAde

aABe

s

These reductions trace out the following rightmost derivation in reverse.

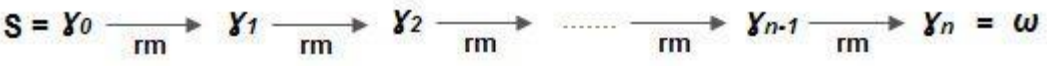


Handle Pruning

• If A −−>β is a production then reducing βto A by the given production is called handle pruning i.e., removing the children of A from the parse tree.

• A rightmost derivation in reverse can be obtained by *handle pruning.*

Start with a string of terminals *ω* that is to parse. If *ω* is a sentence of the grammar at hand, then *ω*= γ_n where γ_n is the *nth* right sentential form of some as yet unknown rightmost derivation.



Example for right sentential form and handle for grammar

E −−> E+E

E −−> E*E

E −−> (E)

E −−> **id**

Right sentential form	Handle	Reduction production
id_1+ id_2 * id_3	id_1	E --> id
E + id_2 * id_3	id_2	E --> id
E + E * id_3	Id_3	E --> id
E + E * E	E * E	E --> E * E
E + E	E + E	E --> E + E
E		

Shift-reduce Parsing

i) Shift Reduce parsing is a bottom-up parsing that reduces a string *w* to the start symbol of grammar.

ii) It scans and parses the input text in one forward pass without backtracking.

Stack implementation of shift-reduce parsing

• Handle pruning must solve the following two problems to perform parsing:

  o Locating the substring to be reduced in a right sentential form.

  o Determining what production to choose in case there is more than one productions with that substring on the right side.

• The type of data structure to use in a shift-reduce parser.

Implementation of shift-reduce parser

Shift-reduce parser can be implemented by using the following components:

• Stack is used to hold grammar symbols.

• An input buffer is used to hold the string *w* to be parsed.

• $ is used to mark the bottom of the stack and also the right end of the input.

• Initially the stack is empty and the string *ω*is on the input, as follows:

Stack	Input
$	*ω $*

• The parser processes by shifting zero or more input symbols onto the stack until a handle β is on top of the stack.

• The parser then reduces β to the left side 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	$

• When the input buffer reaches the end marker symbol $ and the stack contains the start symbol, the parser halts and announces successful completion of parsing.

Actions in shift-reduce parser

A shift-reduce parser can make four possible actions viz: 1) shift 2) reduce 3) accept 4) error.

• A *shift* action, shifts the next symbol onto the top of the stack.

• A *reduce* action, replaces the symbol on the right side of production by the symbol on left side of the production concerned.

To perform reduction, the parser must know the right end of the handle which is at the top of the stack. Then the left end of the handle within the stack is located and the non-terminal to replace the handle is decided.

• An *accept* action, initiates the parser to announce successful completion of parsing.

• An *error* action, discovers that a syntax error has occurred and calls an error recovery routine.

Note:

An important fact that justifies the use of a stack in shift-reduce parsing is that the handle will always appear on top of the stack and never inside.

(eg.) Consider the grammar

E −−> E+E

E −−> E*E

E −−> (E)

E −−> **id**

and the input string id_1+ id_2 * id_3**.** Use the shift-reduce parser to check whether the input string is accepted by the above grammar.