TURING MACHINE

DEFINITION

A Turing machine M is a 7-tuple, namely (Q, ∑, ┌, δ, qo. b, F), where

1. **Q is a finite nonempty set of states.**
2. **┌ is a finite nonempty set of tape symbols**
3. **b ϵ ┌ is the blank.**
4. **∑ is a nonempty set of input symbols and is a subset of ┌ and b ϵ:∑**
5. **δ is the transition function mapping (q, x) onto (q', y, D) where D denotes the direction of movement of R!W head: D =L or R according as the movement is to the left or right.**
6. **6. qo E Q is the initial state**
7. **7. F C Q is the set of final states.**

REPRESENTATION OF TURING MACHINES

(i) Instantaneous descriptions using move-relations.
(ii) Transition table. And
(iii) Transition diagram (transition graph).

INSTANTANEOUS DESCRIPTIONS OF TURING MACHINE

An ID of a Turing machine M is a string αßγ, where ß is the present state of M, the entire input string is split as αγ, the first symbol of γ is the current symbol α under the R/W head and γ has all the subsequent symbols of the input string, and the string α is the substring of the input string formed by all the symbols to the left of α.

TYPES OF TURING MACHINE

1. Multiple track Turing Machine:

- **A k-tack Turing machine(for some k>0) has k-tracks and one R/W head that reads and writes all of them one by one.**
- **A k-track Turing Machine can be simulated by a single track Turing machine**

2. Two-way infinite Tape Turing Machine:

- **Infinite tape of two-way infinite tape Turing machine is unbounded in both directions left and right.**
- **Two-way infinite tape Turing machine can be simulated by one-way infinite Turing machine(standard Turing machine).**

3. Multi-tape Turing Machine:

- **It has multiple tapes and controlled by a single head.**
- **The Multi-tape Turing machine is different from k-track Turing machine but expressive power is same.**
- **Multi-tape Turing machine can be simulated by single-tape Turing machine.**

4. Multi-tape Multi-head Turing Machine:

- **The multi-tape Turing machine has multiple tapes and multiple heads**
- **Each tape controlled by separate head**
- **Multi-Tape Multi-head Turing machine can be simulated by standard Turing machine.**

5. Multi-dimensional Tape Turing Machine:

- **It has multi-dimensional tape where head can move any direction that is left, right, up or down.**
- **Multi dimensional tape Turing machine can be simulated by one-dimensional Turing machine**

6. Multi-head Turing Machine:

- **A multi-head Turing machine contains two or more heads to read the symbols on the same tape.**
- **In one step all the heads sense the scanned symbols and move or write independently.**
- **Multi-head Turing machine can be simulated by single head Turing machine.**

7. Non-deterministic Turing Machine:

- **A non-deterministic Turing machine has a single, one way infinite tape.**
- **For a given state and input symbol has atleast one choice to move (finite number of choices for the next move), each choice several choices of path that it might follow for a given input string.**
- **A non-deterministic Turing machine is equivalent to deterministic Turing machine**

8. Universal Turing Machine:

- **A Turing machine is said to be universal Turing machine if it can accept:**
  - **The input data, and**

- **An algorithm (description) for computing.**

PARSING

In a natural language, parsing is the process of splitting a sentence into words. There are two types of parsing, namely the top-down parsing and the bottomup parsing. Suppose we want to parse the sentence "Ram ate a mango." If NP, VP, N. V, ART denote noun predicate, verb predicate. noun. verb and article, then the top-down parsing can be done as follows:

S -> NPVP
-> Name VP
-> Ram V NP
-> Ram ate ART N
-> Ram ate a N
-> Ram ate a mango

The bottom-up parsing for the same sentence is

Ram ate a mango -> Name ate a mango
-> Name verb a mango
-> Name V ART N
-> NP VN P
-> NP VP

TYPES OF PARSING

1. **Top Down Parsing**
2. **Bottom Up Parsing**

TOP DOWN PARSING

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder. Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string

AMBIGUITY

A grammar that produces more than one parse tree for some sentence is said to be ambiguous.

ELIMINATING LEFT RECURSION

A grammar is left recursive if it has a nonterminal A such that there is a derivation for some string $A =>^{+} A\alpha$. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. The left-recursive pair of productions

$A \rightarrow A\alpha$

could be replaced by the non-left-recursive productions:

$A \rightarrow \beta A'$

$A' \rightarrow \alpha A' | \epsilon$

without changing the strings derivable from A.

ELIMINATING LEFT FACTORING

In general, if $A \rightarrow \alpha \beta_1 | \alpha \beta_2$ are two A-productions, and the input begins with a nonempty string derived from α_1 we do not know whether to expand A to $\alpha \beta_1$ or $\alpha \beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α ,we expand A' to β_1 or β_2. That is, left-factored, the original productions become

$A \rightarrow \alpha A'$

$A' \rightarrow \alpha \beta_1 | \alpha \beta_2$

FIRST and FOLLOW

FIRST

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

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

2. If X is a nonterminal and $X \rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in $FIRST(Y_i)$, and ϵ is in all of $FIRST(Y_1),\ldots,FIRST(Y_{i-1})$; that is, $Y_1 \ldots Y_{i-1}) =>^{*} \epsilon$. If ϵ is in $FIRST(Y_j)$ for all j =1,2,…,k, then add ϵ to FIRST(X). For example, everything in $FIRST(Y_1)$ is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 =>^{*} \epsilon$, then we add $FIRST(Y_2)$, and so on.

3. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).

FOLLOW

1. Place $ in FOLLOW(S), where S is the start symbol, and $ is the input right endmarker.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ in FOLLOW(B).

3. If there is a production A->α B , or a production A->α B ß , where FIRST(ß) contains ϵ , then everything in FOLLOW(A) is in FOLLOW(B).

LL(1)
It is a Non Recursive Decent Parsing.Here the 1st L represents that the scanning of the Input will be done from Left to Right manner and the second L shows that in this parsing technique we are going to use Left most Derivation Tree. And 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.

ALGORITHM TO CONSTRUCT LL(1) PARSING TABLE:

Step 1: First check for left recursion in the grammar, if there is left recursion in the grammar remove that and go to step 2.

Step 2: Calculate First() and Follow() for all non-terminals.

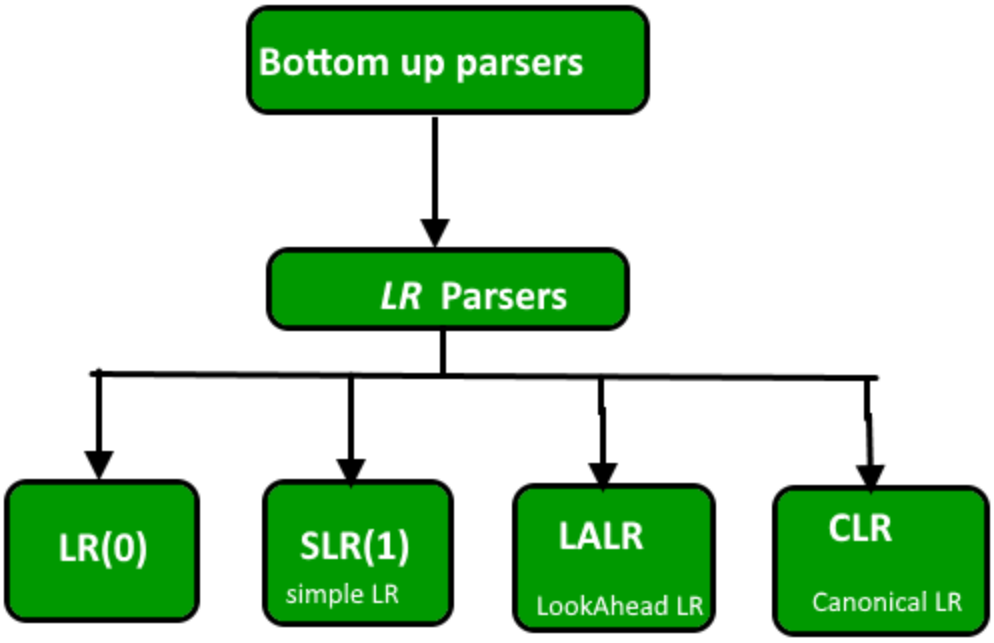
1. **First(): If there is a variable, and from that variable, if we try to drive all the strings then the beginning Terminal Symbol is called the First.**
2. **Follow(): What is the Terminal Symbol which follows a variable in the process of derivation.**

Step 3: For each production A –> α. (A tends to alpha)

1. **Find First(α) and for each terminal in First(α), make entry A –> α in the table.**
2. **If First(α) contains ε (epsilon) as terminal than, find the Follow(A) and for each terminal in Follow(A), make entry A –> α in the table.**
3. **If the First(α) contains ε and Follow(A) contains $ as terminal, then make entry A –> α in the table for the $.**

BOTTOM UP PARSING (SHIFT REDUCE PARSING)

Build the parse tree from leaves to root. Bottom-up parsing can be defined as an attempt to reduce the input string w to the start symbol of grammar by tracing out the rightmost derivations of w in reverse.



A general shift reduce parsing is LR parsing. The L stands for scanning the input from left to right and R stands for constructing a rightmost derivation in reverse.

Benefits of LR parsing:

1. **Many programming languages using some variations of an LR parser. It should be noted that C++ and Perl are exceptions to it.**
2. **LR Parser can be implemented very efficiently.**
3. **Of all the Parsers that scan their symbols from left to right, LR Parsers detect syntactic errors, as soon as possible.**

LR(0)

We need two functions

1. **Closure()**
2. **Goto()**

Augmented Grammar

If G is a grammar with start symbol S then G', the augmented grammar for G, is the grammar with new start symbol S' and a production S' -> S. The purpose of this new starting production is to indicate to the parser when it should stop parsing and announce acceptance of input.

LR(0) Items

An LR(0) is the item of a grammar G is a production of G with a dot at some position in the right side.

Closure Operation

If I is a set of items for a grammar G, then closure(I) is the set of items constructed from I by the two rules:

1. **Initially every item in I is added to closure(I).**
2. **If A -> α.Bβ is in closure(I) and B -> γ is a production then add the item B -> .γ to I, If it is not already there. We apply this rule until no more items can be added to closure(I).**

Construction of GOTO graph-

- **State I_0 – closure of augmented LR(0) item**
- **Using I_0 find all collection of sets of LR(0) items with the help of DFA**
- **Convert DFA to LR(0) parsing table**

Construction of LR(0) parsing table:

- **The action function takes as arguments a state i and a terminal a (or $, the input end marker). The value of ACTION[i, a] can have one of four forms:**
    1. **Shift j, where j is a state.**
    2. **Reduce A -> β.**
    3. **Accept**
    4. **Error**

SLR(1)

The SLR parser is similar to LR(0) parser except that the reduced entry. The reduced productions are written only in the FOLLOW of the variable whose production is reduced.

Construction of SLR parsing table –

1. **Construct C = { I_0, I_1, ……. I_n}, the collection of sets of LR(0) items for G'.**
2. **State i is constructed from Ii. The parsing actions for state i are determined as follow :**
   - **If [A -> ?.a?] is in I_i and GOTO(I_i , a) = I_j , then set ACTION[i, a] to "shift j". Here a must be terminal.**
   - **If [A -> ?.] is in I_i, then set ACTION[i, a] to "reduce A -> ?" for all a in FOLLOW(A); here A may not be S'.**
   - **Is [S -> S.] is in I_i, then set action[i, $] to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not SLR.**
3. **The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(I_i , A) = I_j then GOTO [i, A] = j.**
4. **All entries not defined by rules 2 and 3 are made error.**

LR(1)

Closure Operation

Closure(I)

repeat

for (each item [A -> ?.B?, a] in I)

for (each production B -> ? in G')

for (each terminal b in FIRST(?a))

add [B -> .? , b] to set I;

until no more items are added to I;

return I;

Goto Operation

Goto(I, X)

Initialise J to be the empty set;

for (each item A -> ?.X?, a] in I)

Add item A -> ?X.?, a] to se J; /* move the dot one step */

return Closure(J); /* apply closure to the set */

Construction of GOTO graph

- **State I_0 – closure of augmented LR(1) item.**
- **Using I_0 find all collection of sets of LR(1) items with the help of DFA**
- **Convert DFA to LR(1) parsing table**

Construction of CLR parsing table

Input – augmented grammar G'

1. **Construct C = { I_0, I_1, ……. I_n} , the collection of sets of LR(0) items for G'.**
2. **State i is constructed from Ii. The parsing actions for state i are determined as follow : i) If [A -> ?.a?, b] is in I_i and GOTO(I_i , a) = I_j, then set ACTION[i, a] to "shift j". ii) If [A -> ?. , a] is in I_i , A ≠ S, then set ACTION[i, a] to "reduce A -> ?". iii) Is [S -> S., $] is in I_i, then set action[i, $] to "accept". If any conflicting actions are generated by the above rules we say that the grammar is not CLR.**
3. **The goto transitions for state i are constructed for all nonterminals A using the rule: if GOTO(I_i, A) = I_j then GOTO [i, A] = j.**
4. **All entries not defined by rules 2 and 3 are made error.**

LALR

LALR parser are same as CLR parser with one difference. In CLR parser if two states differ only in lookahead then we combine those states in LALR parser. After minimisation if the parsing table has no conflict that the grammar is LALR also.

Important Notes

1. Even though CLR parser does not have RR conflict but LALR may contain RR conflict.
2. If number of states LR(0) = n1,number of states SLR = n2,number of states LALR = n3,number of states CLR = n4 then, n1 = n2 = n3 <= n4

3.LR(1) parsers are more powerful parser