LR Parser

Dr. Meera Thapar Khanna

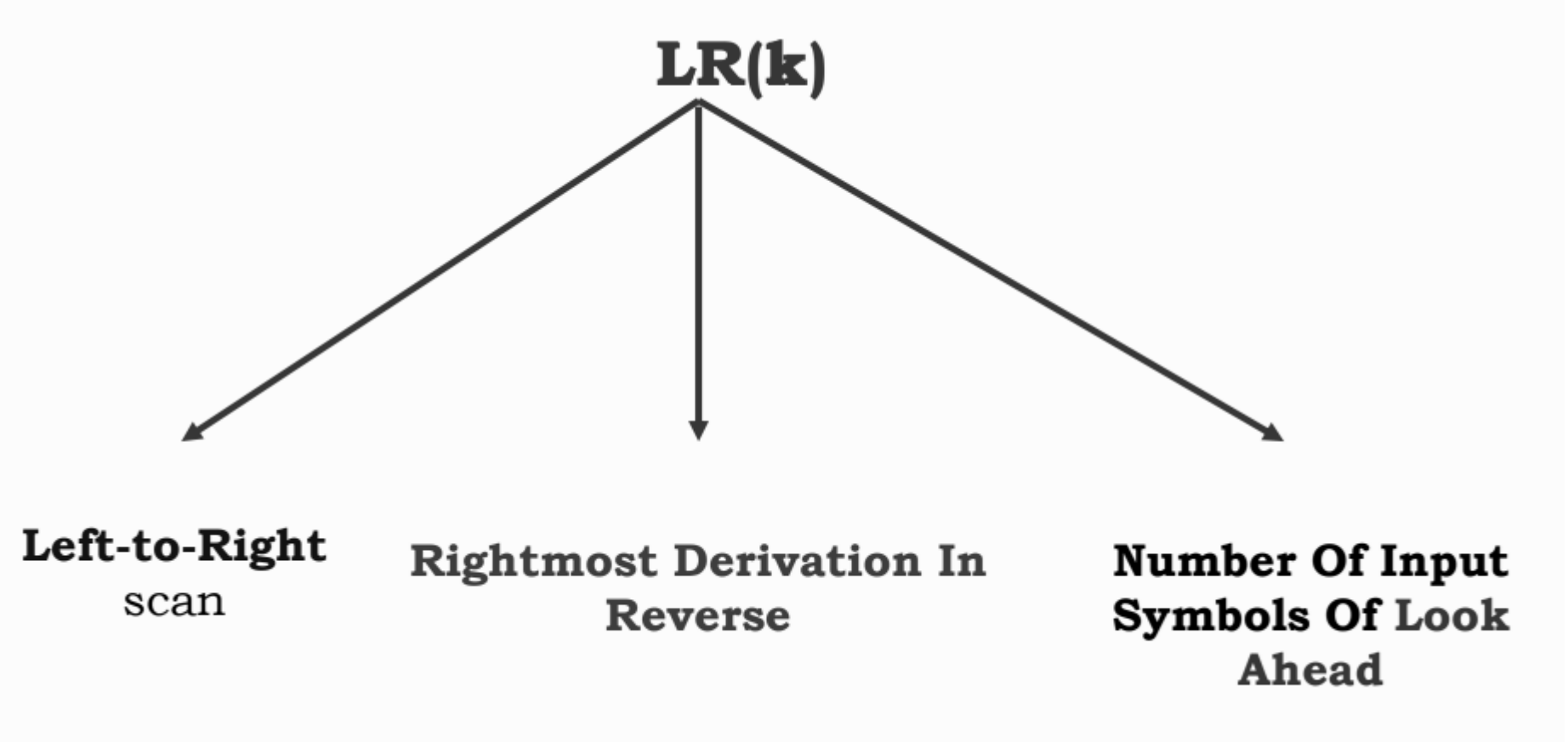
CSE Department

Pandit Deendayal Energy University

LR Parsing

- LR(k) parsing is a type of bottom up parsing. Where, L is left to right scan of the given input string, R is Right Most derivation in reverse and K is no of input symbols as the Look ahead.
- When k is omitted, k is assumed to be 1.

LR(k) Parser


LR(k)
Left-to-Right
scan
Rightmost Derivation In
Reverse
Number Of Input
Symbols Of Look
Ahead

Why LR Parsers

- It is the most general non back tracking shift reduce parsing method.
- The class of grammars that can be parsed using the LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- 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.

Drawback of LR Parser

- It is too much work to construct LR parser by hand for a programming language grammar. A specialized tool, called a LR parser generator is needed.

Types of LR Parser

1. SLR – Simple LR

   Easier to implement, least powerful.

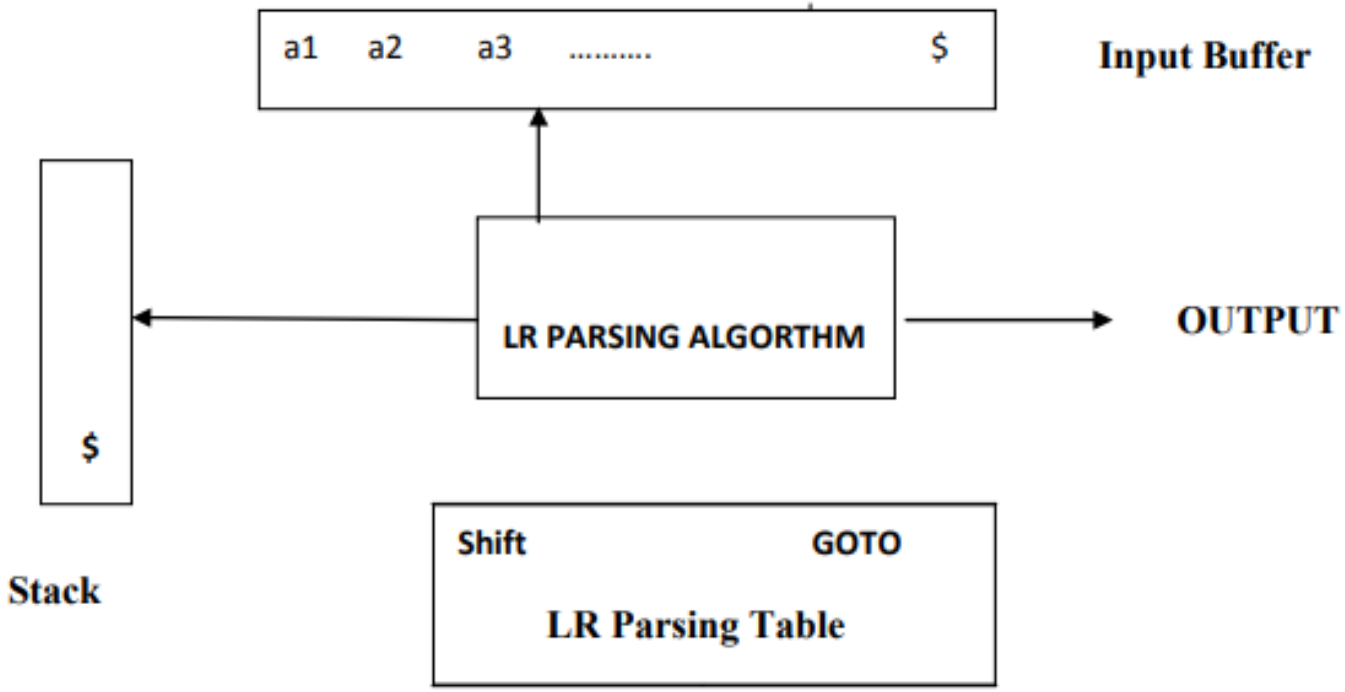
2. CLR – Canonical LR

   Most powerful, most expensive.

3. LALR – Look-Ahead LR

   Intermediate in size and cost between the other two methods.

Components of LR Parsers



Components of LR Parsers

- An input buffer that contains the string to be parsed followed by a $ Symbol, used to indicate end of input.
- A stack containing a sequence of grammar symbols with a $ at the bottom of the stack, which initially contains the Initial state of the parsing table on top of $.

Parse Table

- A parsing table (M), it is a two dimensional array M[state, terminal or Non terminal] and it contains two parts:

1. A Parsing Action function
2. A GOTO function

The Action Part

- The Action Table specifies the actions of the parser (e.g., shift or reduce), for the given parse state and the next token
  - Rows are State Names;
  - Columns are Terminals
- An action table entry can have one of following four kinds of values in it:
  1. Shift X, where X is a State number.
  2. Reduce X, where X is a Production number.
  3. Accept, signifying the completion of a successful parse.
  4. Error entry.

GOTO Table

- The GOTO table specifies which state to put on top of the stack after a reduce.
  - Rows are State Names;
  - Columns are Non-Terminals
- The GOTO Table is important to find out the next state after every reduction.
- The GOTO Table is indexed by a state of the parser and a Non Terminal (Grammar Symbol) ex : GOTO[S, A]
- The GOTO Table simply indicates what the next state of the parser if it has recognized a certain Non Terminal.

Augmented Grammar

- If G is a Grammar with Start Symbol S, the Augmented Grammar G’ is G with a New Start Symbol S’, and New Production S’ →S$.

- The Purpose of the Augmented Grammar is to indicate to the parser when it should stop parsing and announce acceptance of the input

LR(0) Items

- An LR(0) Item of a Grammar G is a Production of G with a Dot ( ) at some position of the right side.

The production A → XYZ yields the Four items:

- A→•XYZ We hope to see a string derivable from XYZ next on the input.
- A→X•YZ We have just seen on the input a string derivable from X and that we hope next to see a string derivable from YZ next on the input.
- A→XY•Z
- A→XYZ•

- The production A→ ε generates only one item,

  A→•.
- Each of this item is a Viable prefixes.

- Closure Item : An Item created by the closure operation on a state.
- Complete Item : An Item where the Item Dot is at the end of the RHS.

Closure Operation

If I is an initial State, then the Closure (I) is constructed as follows:

- Initially, add Augment Production to the state and check for the • symbol in the Right hand side production, if the • is followed by a Non terminal then Add Productions which are Starting with that Non Terminal in the State I.
- If a production X -> α•Aβ is in I and A -> γ is a production, then add Production A -> .γ in the State I. Rule 2 is applied until no more productions added to the State I(meaning that the • is followed by a Terminal symbol).

Example

Grammar

E -> E + T | T

T -> T * F | F

F -> (E) | id

G:

E -> E + T	(1)
E -> T	(2)
T -> T * F	(3)
T -> F	(4)
F -> (E)	(5)
F -> id	(6)

Step1

Grammar Augumentation G‘:

E‘ -> E	(0)
E -> E + T	(1)
E -> T	(2)
T -> T * F	(3)
T -> F	(4)
F -> (E)	(5)
F -> id	(6)

LR(0) items

Closure of E’ -> .E = i0

i(0): E’-> .E

E -> .E + T

E -> .T

T -> .T * F

T -> .F

F -> .(E)

F -> .id

Step 3

GOTO(i0,E) = i1

- E’ -> E.
- E -> E.+T //as dot crosses E

GOTO(i0,T) = i2

- E -> T.
- T -> T.*F

GOTO(i0, F) =i3

- T -> F. //Rule completed

GOTO(i0, () = i4

- F -> (.E) // dot is prefixed to E
- E -> .E+T
- E -> .T //dot is prefixed to T
- T -> .T*F
- T -> .F
- F -> .(E)
- F -> .id

GOTO(i0, id) = i5

- F -> id. //rule completed

Processing of step i0 is completed.

Continue with i1

GOTO(i1, +) = i6

- E -> E+.T
- T -> .T*F
- T -> .F
- F ->.(E)
- F -> .id

GOTO(i2, *) = i7

- T -> T*.F
- F -> .(E)
- F -> .id

Goto(i4, E) = i8

- F -> (E.)
- E -> E.+T

Goto(i4, T)=i2

- E -> T.
- T -> T.*F

Goto(i4, F)=i3

- T -> F.

Goto(i4, ( )=i4

- F -> (.E) // dot is prefixed to E
- E -> .E+T
- E -> .T //dot is prefixed to T
- T -> .T*F
- T -> .F
- F -> .(E)
- F -> .id

Goto(i4,id) =i5

- F -> id.

Goto(i6,T) = i9

- E -> E+T.
- T -> T.*F

Goto(i6,F)=i3

- T -> F.

Goto(i6,( )=i4

- F -> (.E)
- E -> .E+T
- E -> .T //dot is prefixed to T
- T -> .T*F
- T -> .F
- F -> .(E)
- F -> .id

Goto(i6, id) = i5

- F -> id.

Goto(i7,F)=i10

- T -> T*F.

Goto(i7,id) =i5

- F-> id.

Goto(i7,( )=i4

- F -> (.E) // dot is prefixed to E
- E -> .E+T
- E -> .T //dot is prefixed to T
- T -> .T*F
- T -> .F
- F -> .(E)
- F -> .id

Goto(i8,))=i11

- F -> (E).

Goto(i8,+)=i6

- E -> E+.T
- T -> .T*F
- T -> .F
- F ->.(E)
- F -> .id

Goto(i9,*)=i7

- T -> T*.F
- F -> .(E)
- F -> .id

Example

Grammar

S -> AA

A -> aA | b